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

Vincent Rahli
Cornell University

David Guaspari
ATC-NY

Mark Bickford
Cornell University & ATC-NY

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.

1. Introduction

Specification, Verification, and Protocol Synthesis. Formal methods tools such as type checkers, SMT solvers, model checkers, theorem provers, etc., are more and more widely used to develop reliable concurrent systems. Many programming languages with rich type systems (featuring, e.g., dependent, refinement, or session types) have been developed to specify, enforce, and verify both correctness and security properties of concurrent programs [7, 18, 19, 30, 33, 38] and system components in general [24, 37]. These expressive type systems are the basis for specification languages that enable protocol verification and synthesis [6, 11].

We have invested in this line of formal work for several years since there is good evidence that appropriate formal methods can substantially improve the reliability of distributed protocols and that such methods are more necessary for this kind of programming because of its intrinsic complexity. We are also fascinated by this apparent intrinsic complexity—is it real and if so, what are the root causes and appropriate remedies? We think that attributes of specification and proof process are revealing answers to these questions, and we will touch on the possibilities as we proceed.

We use *constructive logic* because it supports *provably correct* code synthesis from proofs and because distributed computing has aspects which are essentially constructive in that the agents must have concrete information to make decisions, and the program specifications are about how to acquire that information. We say "provably correct" because program correctness is guaranteed by machine checked proofs that these programs satisfy desired correctness properties.

One reason that distributed systems are especially difficult to code correctly and maintain is that they are difficult to reason about. This does not become clear from model checking alone nor from testing because those methods do not expose the reasoning need to make definitive arguments. Once the right proof methods are developed, we can see the complexity of the reasoning task exposed in the size of the proof and the tight connections between its components.

We also see why formal methods tools can be a great help to programmers to guarantee code correctness, and even though verifying systems correctness using proof assistants is more time consuming, it does provide very strong correctness guarantees and insights into the programming task. Those insights have led us to the *event combinators* we describe here. Moreover as Klein et al. write: "Complete formal verification is the only known way to guarantee that a system is free of programming errors" [24].

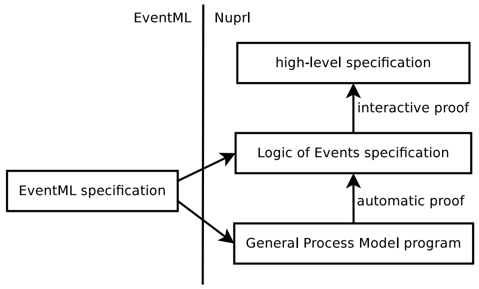
We use the EventML language to specify protocols. EventML works synergistically with the Nuprl proof assistant [4, 14, 25] which is closely related to the Coq [1, 5] proof assistant. Nuprl is a programing/logical environment based on Constructive Type Theory (CTT) [4, 14] and on Classic ML [21], that allows one to prove mathematical results but also to 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* that allow programmers 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: the Logic of Events (LoE) [8, 10, 12] to specify and reason about the information flow of distributed programs, as well as a General Process Model (GPM) [9] to implement these information flows. The semantic meaning of an EventML program can be expressed both by a LoE formula and by a GPM program. This dualism provides a workable balance between programming and formal reasoning. Because of this dualism we also refer to EventML programs as "constructive specifications".

Currently, EventML *docks* with the Nuprl logical programming environment. Since every EventML type is a Nuprl type, docking means that any Nuprl object whose type is an EventML type can be imported into an EventML program. This includes not only operations from Nuprl's mathematical library, but also Nuprl objects denoting executable processes.

In the other direction, programmers can specify a protocol in EventML and upload it to a theorem prover to verify its properties. Nuprl is well suited to reason about distributed systems because of its constructive logic, its expressiveness, and its large library of proven facts about verified programs and about the Logic of Events. But, in principle, EventML can connect to any prover that implements the Logic of Events and our General Process Model [9].

The diagram presented below summarizes the interaction between EventML and Nuprl. The advantage of using EventML is that it greatly facilitates the developing, testing, and debugging of distributed protocols. Once we have extracted the semantic meaning of an EventML specification in terms of a LoE formula and GPM program, we automatically prove that the program satisfies the formula. It then remains to interactively prove that the LoE formula satisfies the desired correctness properties.



Our Computation Model. EventML's computation model is based on the General Process Model [9]. 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):

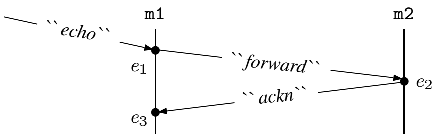
$$\texttt{Proc} = \texttt{corec}(\lambda P.(A \rightarrow P \times \texttt{Bag}(B)) + \texttt{Unit})$$

`Unit` is a singleton type and $+$ is the disjoint union type. Intuitively, this definition of `Proc` says that a process is one of two things: a function that can accept an input of type A, generate a (possibly empty) bag of output values of type B, and become 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. Processes can then be executed by interpreting them according to the evaluation rules of Nuprl's computation system.

The Logic of Events. The Logic of Events [8, 10], related to Lamport's notion of causal order [26], was developed to reason about events occurring in the execution of a distributed system. In the context of this paper, an event is an abstract object 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 (i.e., a temporal ordering). The Logic of Events describes systems in terms of the causal relations among events and (ultimately) their primitive information. It abstracts away from low-level details, such as setting up communication channels, unless they are essential to the protocol.

To reason about a protocol in the Logic of Events, we reason about its possible runs. An *event ordering* is an abstract representation of one run of a distributed system; it is a structure consisting of: a set of events; a function `loc` that associates a *location* with each event; a function `info` that associates primitive information with each event; and a well-founded *causal ordering* relation, $<$, on events [26]. We express system properties as predicates on event orderings. A system satisfies such a property if every execution satisfies the predicate.

The following message sequence diagram depicts a simple event ordering:



Event e_1 corresponds to the receipt of a message with header ``*echo*`` by machine `m1`. Upon receipt of that ``*echo*`` message, `m1` forwards it to `m2`, which causes event e_2. Upon receipt of that ``*forward*`` message, `m2` sends an acknowledgment to `m1`, which causes event 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 $e_1 <_{\texttt{loc}} e_3$ (for any two events e_1 and e_2, $e_1 <_{\texttt{loc}} e_2$ is defined as $e_1 < e_2 \wedge \texttt{loc}(e_1) = \texttt{loc}(e_2)$).

Event Classes. A basic concept in the Logic of Events is that of an *event class* [8], which effectively partitions the events of an event ordering into those it "recognizes" and those it does not, and associates values to the events it recognizes. They can therefore be regarded as combinations of event recognizers and event handlers. For example, the *base class* called `vote'base` will recognize the arrival of any message with header ``*vote*`` and handle that event by simply returning the content of the message[2]. Another class, call it `X`, may recognize that, in the context of a particular run of some protocol, the arrival of a ``*vote*`` message signifies successful completion and will assign to such an event a value that means "send the 'success' message." `X` will recognize some but not all ``*vote*`` events; and these two classes will assign different values to the events they both recognize. Therefore, event classes classify events not only in terms of input messages but also in terms of the outputs they compute from these input messages. They classify events not only in terms of the ones they recognize but also in terms of how they handle them. We specify systems in EventML by defining and combining such event recognizers and handlers that appropriately classify system events.

Event classes specify information flow in a network of reactive agents by *observing* the information computed by the agents when events occur—i.e., on receipt of messages. It turns out that event classes form a monad [32] encapsulating local computations in distributed systems. One reasons about a specification by reasoning about the observations made by its components. As mentioned above, some event classes of the Logic of Events, such as the ones described in EventML specifications, are implementable by GPM processes. These classes can be seen as processes that aggregate information from input messages and past observations into an internal state and compute appropriate outputs. In context, we will use whichever terminology seems to us most intuitive (class/observer/process).

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

[2] In general, a base class recognizes the arrival of a particular kind of message (identified by its header) and observes its body.

Formally, an event class is a function whose inputs are an event ordering and an event, and whose output is a bag (multiset) of values (observations). If the values are of type T, the class is called an event class of type T. The collection of all event classes of type T is the type $\texttt{Class}(T) = \texttt{EO} \rightarrow \texttt{E} \rightarrow \texttt{Bag}(T)$, where `EO` is the type of event orderings and `E` the type of events in a given event ordering. For example, the following event class of type `Class(Loc)`, where `Loc` is the location type, recognizes any event and observes its location: $\lambda eo.\lambda e.\{\texttt{loc}(e)\}$. We reason about event classes in terms of the *event class relation*, which relates events, observers, and observations: we say that the event class 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 $(X \ eo \ e)$. Our notation omits eo because the choice of eo is irrelevant to the present discussions. If the bag is nonempty we say that e is an X-event.

An EventML specification describes event classes that produce and consume messages (among other values), and especially, it describes a *main class* that specifies the entire information flow of a system. Main classes observe *directed messages* represented by pairs location/message. Given such a directed message (loc, msg), the communication system attempts to deliver message msg to location loc. This directed message can be seen as the instruction "send message msg to location loc". We reason about the behavior of a specification based on assumptions about message delivery. 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 [21], 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 class in terms of the event class relation. It characterizes the response to any event e in terms of observations 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.

Moreover, 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 and shows how it can be used to define a fault-tolerant consensus protocol (Section 2) and how one can exploit the connection to Nuprl to prove its safety properties (Section 3) and generate a verified implementation (Section 4).

2. A Specification of 2/3 Consensus

Consider the following problem: A system has been replicated for fault tolerance [36]. It responds to *commands* identified by values in some type `Cmd`, a parameter of the specification. Commands are issued to any of the *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, therefore given the FLP impossibility result [15], consensus might never be reached. We assume that all failures are crash failures: that is, a failed replica ceases all communication with its surroundings. The 2/3 consensus protocol tolerates up to `flrs` failures (also a parameter of the specification), by using precisely $3 * \texttt{flrs} + 1$ replicas. An appealing feature of the protocol is that with a small change, and using $5 * \texttt{flrs} + 1$ replicas, it can tolerate Byzantine failures.

Input events communicate *proposals*, which consist of slot number/command pairs, where slot numbers are modeled by integers: (n, c) proposes that command c be the n^{th} one performed. The protocol is intended to obtain agreement, for each n, on which command will be the n^{th} to be performed, and to broadcast `notify` messages with those decisions to a list of *clients* (also a parameter of the specification).

Each copy of the replicated system contains a module that carries out the consensus negotiations. In this specification we describe only those modules (which we continue to call *Replicas*). To save space, we omit an EventML description of how these consensus decisions are used. An account of that may be found in the description of the Paxos protocol in [35].

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 classes that act like state machines. Section 2.2 describes one of those state machines, which performs the key computation used to detect consensus. Section 2.3 shows how EventML defines an event class to accomplish that. Figures 1 and 2 provide the full EventML specification.

Interface. An input event to the protocol is the arrival of a message that communicates a proposal. The header of such a message is ``*propose*`` and its data has type `Proposal`, which we declare as follows:

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

The data (n, c) of a ``*propose*`` message proposes that command c be the n^{th} one performed. The command type is defined as a parameter of the specification as follows:

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

One subtlety: in addition to defining the type parameter `Cmd`, this declaration also supplies an "equality decider", which we call `cmdeq`—an operation that determines whether two values of `Cmd` are equal. The ability to compare commands is necessary to decide whether or not consensus has been reached. We declare the `propose` interface to the protocol as follows:

```
input propose : Proposal
```

which implicitly defines the base class, called `propose'base`, that detects these input events, i.e., input messages with headers ``*propose*``, and observes their data.

To characterize the top-level actors in the protocol we define the event class `Replica` below. Its outputs are directed messages with header ``*notify*``; the data of such a message identifies a proposal that has achieved consensus:

```
output notify : Proposal
```

This interface definition does not declare a base class recognizing the arrival of ``*notify*`` messages; those events occur outside the system. However, it implicitly declares the two functions `notify'send` and `notify'broadcast`, which are convenient operations used to send messages. Let msg be the ``*notify*`` message with body p. Then the expression $(\texttt{notify'send} \ l \ p)$ is (l, msg), the directed message instructing that msg be sent to l; and $(\texttt{notify'broadcast} \ \{l_1, l_2, \ldots\} \ p) = \{(l_1, msg), (l_2, msg), \ldots\}$, which is a bag 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, which are used by the replicas to cast their votes to the collection of replicas; ``*decided*`` messages which are used to inform the collection of replicas that consensus has been detected on a particular proposal; and ``*retry*`` messages which are described below.

Replicas. 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 a class that responds only to events at the locations in `reps`, or that it is the result of installing an "instance" of `Replica` at each of those locations. The "`@`" idiom guarantees that `SC` will only be installed at a finite number of locations, and is therefore implementable by a finite distributed system.

For each n, the protocol conducts a separate election to vote on proposals for the n^{th} command. We define `Replica` so that it acts simply by spawning subprocesses that cast votes in these elections and identify the winners. The spawning (delegation) operator "`_>>=_`" is an EventML primitive:[3]

```
class Replica = NewVoters >>= Voter ;;
```

The event class `NewVoters` decides when to spawn a new voting process. `Voter` is a higher-type function; the values it returns are event classes that do the voting. When some `NewVoters`-event e occurs and $v \in$ `NewVoters`(e), `Replica` spawns a *local instance* of the class (`Voter` v).

By local instance we mean this: When a `NewVoters`-event e occurs at location *loc*, the subprocess spawned acts only at *loc*. The only messages it can react to are messages arriving at *loc* (and, of course, only those arriving 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" classes.)

A note on terminology: The specification will define several higher-order functions, such as `Voter`, that return event classes. For convenience we will use "a `Voter` class" or "a `Voter`" as a shorthand for "an event class returned by `Voter`."

`NewVoters` acts like a state machine. More precisely, it defines a distinct state machine at each location, each reacting to inputs at that location. Its inputs are messages either with header ``*proposal*`` (coming from outside the system) or with header ``*vote*`` (coming from within the system). If an input at location *loc* is a proposal (n, c) or a vote for proposal (n, c) and if, in addition, location *loc* has not previously received a proposal or vote for the n^{th} command, that input is a `NewVoters`-event; and it will cause (`NewVoters >>= Voter`) to spawn a local instance of the event class (`Voter` (n, c)) at location *loc*, which will start voting for (n, c).

Voters. `Voter` classes send votes and tally the votes they receive to determine whether some proposal has achieved consensus. We will not allow a `Voter` to announce a consensus for proposal (n, c) unless it has received $2 *$ `flrs` $+ 1$ votes for (n, c) from $2 *$ `flrs` $+ 1$ different replicas.

We cannot guarantee that any particular poll of the `Voter` classes 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 *rounds*. A pair (n, r)—(slot number/round number)—is called a *voting round* (or just round for short). Pairs of the form $((n, r), c)$—-(voting round/command)—are called *ballots*. Thus, a `Voter` casts votes not merely for a particular proposal but for a particular proposal in a particular round. Votes are pairs of the form $(((n, r), c), loc)$—(ballot/location). The location indicates the sender of the vote. When a replica R votes, it puts its own location in the message so that the receiver of that vote will know whether or not R has already voted. Replicas ignore duplicate votes. Therefore, the protocol will work even if messages get duplicated.

Using the parallel combinator "`_||_`" we define the `Rounds` event class in charge of doing the successive polls, and the `Voter` event class, which calls its sub-component `Rounds` and sends notifications to the clients of the system once a round achieves consensus:

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

`Rounds`, `Round`, `NewRounds`, and `Notify` are also higher-order functions that return event classes.

A local instance of the event class (`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 `Rounds`' second component, may cast votes for other proposals in later rounds.

`Round` 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. Section 2.2 provides a detailed account of `Round`.

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

(`Notify` n) detects the arrival of a ``*decided*`` message with data (n, c) indicating that consensus has been reached about the n^{th} command, and sends notifications to the clients of the system indicating that slot n has been filled with command c. For any event classes `A` and `B`, the class (`A until B`) acts like `A` until a `B`-event occurs, at which point it terminates. This allows us to terminate any voting for n once consensus has been reached on n.

2.2 Detecting Consensus

The structure so far described is common to many consensus protocols: spawn a separate process to conduct each election; conduct each election in a sequence of rounds that are spawned as needed. What distinguishes one protocol from another is the algorithm that detects consensus. (`Round` $((n, r), c)$) has two components:

```
class Round ((n,r),c) =
  Output(\loc.vote'broadcast 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; we will not explain this further here. 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 a class that acts like `A` but terminates after the first `A`-event, i.e., it terminates as soon as `A` produces something. Because there is at most one (`Quorum` (n, r))-event at any location the use of `Once` is logically redundant; but it effects an optimization because it guarantees that a process is cleaned up once it has produced an output.

[3] We use the symbol ">>=" because delegation is the bind operator in the event class monad.

Figure 1 2/3 consensus: Part 1

```
specification two_thirds

(* ------------ Parameters ------------ *)
(* consensus on commands of arbitrary type Cmd with equality decider cmdeq *)
parameter Cmd, cmdeq : Type * Cmd Deq
parameter flrs       : Int      (* max number of failures           *)
parameter reps       : Loc Bag  (* locations of (3 * flrs + 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               internal vote  : Vote               internal decided : Proposal
output notify  : Proposal               internal retry : Ballot

(* ------------ Quorum: a state machine ------------ *)
(* -- filter -- *)
let newvote (n,r) (((n',r'),cmd),sender) (cmds,locs) = (n,r) = (n',r') & !(deq-member (op =) sender locs);;

(* -- update -- *)
let addvote (((n,r),cmd),sender) (cmds,locs) = (cmd.cmds, sender.locs);;
let add_to_quorum (n,r) loc vt state = if newvote (n,r) vt state then addvote vt state else state;;

(* -- output -- *)
let roundout loc (((n,r),cmd),sender) (cmds,locs) =
  if length cmds = 2 * flrs
  then let (k,cmd') = poss-maj cmdeq (cmd.cmds) cmd in
         if k = 2 * flrs + 1
         then  decided'broadcast reps(n, cmd')
         else { retry'send loc ((n,r+1),cmd') }
  else {} ;;
let when_quorum (n,r) loc vt state = if newvote (n,r) vt state then roundout loc vt state else {} ;;

(* -- state machine -- *)
class QuorumState (n,r) = Memory(\loc.([],[]), add_to_quorum (n,r), vote'base) ;;
class Quorum (n,r) = (when_quorum (n,r)) o (vote'base, QuorumState (n,r)) ;;
```

(`Quorum` (n, r)) produces an output as soon as it has received votes in round (n, r) from $2 * \texttt{flrs} + 1$ 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` event classes and will results in ``*notify*`` messages being sent). If the votes it has received are not unanimous then it is logically possible that, however many more votes are tallied, no proposal will receive $2 * \texttt{flrs} + 1$ votes on this round. (Note that if `flrs` failures have occurred, no more votes will arrive, so the `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 $2 * \texttt{flrs} + 1$ 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` at one location detects a consensus about command n and a `Quorum` at another location calls for a new round of voting about n. So multiple ``*notify*`` messages may be sent about n, in a single round or in different rounds. Section 3 describes a proof that, for any n, all ``*notify*`` messages about the n^{th} command will agree on which command has been chosen.

2.3 Implementing `Quorum`

`Quorum (n,r)` acts like a state machine—more precisely, like a Mealy machine: In response to inputs it may change state and produce outputs. It is convenient to factor its definition. We first define (`QuorumState (n,r)`), a Moore machine that maintains a suitable state, i.e., the collection of votes for round `(n,r)` that the process has received thus far. (`Quorum (n,r)`) will observe (`QuorumState (n,r)`) and issue directed messages as appropriate.

EventML provides primitives for defining Moore machines. We use the primitive `Memory` to define `QuorumState`:

```
class QuorumState (n,r) =
  Memory(\loc.([],[]),
         add_to_quorum (n,r),
         vote'base) ;;
```

A (`QuorumState` (n, r)) Moore machine maintains 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, a `QuorumState` can recognize and 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:

- The expression (`\loc.([],[])`) has type `Loc -> StateType`. It assigns the initial state to each location. In this case, the initial state at any location is a pair of empty lists.
- The transition function (`add_to_quorum(n,r)`) has the following type: `Loc -> InputType -> StateType -> StateType`. It

Figure 2 2/3 consensus: part 2

```
(* ------------ Round ------------ *)
class Round ((n,r),cmd) = Output(\loc.vote'broadcast reps (((n,r),cmd),loc)) || Once(Quorum (n,r)) ;;

(* ------------ NewRounds: a state machine ------------ *)
(* -- inputs -- *)
let vote2retry loc (((n,r),cmd),sender) = {((n,r),cmd)};;
class RoundInfo = retry'base || (vote2retry o vote'base);;

(* -- update -- *)
let update_round n loc ((n',r'),cmd) round = if n = n' & round < r' then r' else round ;;

(* -- output -- *)
let when_new_round n loc ((n',r'),cmd) round = if n = n' & round < r' then {((n',r'),cmd)} else {} ;;

(* -- state machine -- *)
class NewRoundsState n = Memory(\loc.0, update_round n, RoundInfo) ;;
class NewRounds n = (when_new_round n) o (RoundInfo, NewRoundsState n) ;;

(* ------------ Voter ------------ *)
let decision n loc (n',c) = if n = n' then notify'broadcast clients (n,c) else {};;
class Notify n = Once((decision n) o decided'base);;

class Rounds (n,cmd) = Round ((n,0),cmd) || (NewRounds n >>= Round);;
class Voter (n,cmd) = (Rounds (n,cmd) until (Notify n)) || (Notify n);;

(* ------------ NewVoters: a state machine ------------ *)
(* -- inputs -- *)
let vote2proposal loc (((n,round),cmd),sender) = {(n,cmd)} ;;
class Proposal = propose'base || (vote2proposal o vote'base);;

(* -- filter -- *)
let new_proposal (n,cmd) (max,missing) = n > max or deq-member (op =) n missing;;

(* -- update -- *)
let onnewpropose (n,cmd) (max,missing) =
  if n > max then (n, missing ++ (from-upto (max + 1) n))
  else (max, list-diff (op =) missing [n]) ;;
let update_replica (n,cmd) state = if new_proposal (n,cmd) state then onnewpropose (n,cmd) state else state ;;

(* -- output -- *)
let when_new_proposal loc (n,cmd) state = if new_proposal (n,cmd) state then {(n,cmd)} else {} ;;

(* -- state machine -- *)
class ReplicaState = Memory update_replica (\loc.(0,[])) Proposal ;;
class NewVoters = when_new_proposal o (Proposal, ReplicaState) ;;

(* ------------ Replica & Main program ------------ *)
class Replica = NewVoters >>= Voter ;;
main SC where SC = Replica @ reps
```

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 loc, and loc is not listed in the current state, then `add_to_quorum` returns the result of prepending c and loc, respectively, to the components of the current state; otherwise, the current state stays unchanged.

- The event class `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, instead of `Memory`, the primitive combinator `State`.

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

```
class 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)`), of type (`Loc -> InputType -> StateType -> OutputType Bag`), to `loc`(e), and to the values observed at e by `vote'base` and (`QuorumState (n,r)`). Only ``*vote*`` events will be observed by (`Quorum (n,r)`), but not all of them since (`when_quorum (n,r)`) will sometimes return an empty bag. If an input vote arrives for c from loc, and loc is listed in the current state, then `when_quorum` does not output anything. Otherwise, it calls `roundout`, which requires the most complex definition in the specification:

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

The first argument `loc` matches the location at which the input event, that triggered the call to `roundout`, occurs—this event must have a ``*vote*`` message for primitive information; 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 $2 * \texttt{flrs} + 1$ votes; accordingly, the input does not cause an output, so `Quorum` returns an empty bag, and the input event is not a `(Quorum (n,i))`-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 the Nuprl library), implements the Boyer-Moore majority vote algorithm. In the expression

```
let (k,c') = poss-maj cmdeq (c.cmds) c
```

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 * flrs + 1)` therefore tests whether the vote is unanimous. If so, the function returns a bag instructing that the choice of `c'` be broadcast in appropriate ``*decided*`` messages; if not, it returns a bag whose single element is an instruction to send a ``*retry*`` message. Recall that the declaration of ``*retry*`` messages introduces the operation `retry'send`, for constructing directed messages. The expression

```
retry'send loc ((n,r+1),c')
```

is the instruction to send to location `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

Now that we have specified this 2/3 consensus protocol in EventML, we can generate a LoE specification and a GPM program that express the semantic meaning of the EventML specification in our two models of distributed computing. We verify the correctness of the EventML specification 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 Section 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 in Nuprl for the 2/3 consensus protocol of section 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.[4]

Agreement says that notifications never contradict one another:

$$\begin{array}{l}\forall e_1, e_2 : \texttt{E}.\ \forall loc_1, loc_2 : \texttt{Loc}.\ \forall n : \mathbb{Z}.\ \forall c_1, c_2 : \texttt{Cmd}.\\ \quad \left(\begin{array}{l}\ \ (\texttt{notify'send}\ loc_1\ (n, c_1)) \in \texttt{SC}(e_1)\\ \wedge\ (\texttt{notify'send}\ loc_2\ (n, c_2)) \in \texttt{SC}(e_2)\end{array}\right)\\ \quad \Rightarrow\ c_1 = c_2\end{array}$$

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

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

(The reader can think of $\downarrow\exists$ as a classical existential[5].)

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 class 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 $(\texttt{vote'send}\ loc\ v) \in \texttt{SC}(e)$. This can be enforced, e.g., by physical means or by message encryption.

We must also assume a constraint on parameters: `reps` is a bag, without repetitions, of size $3 * \texttt{flrs} + 1$.

3.3 Automation

To help us prove such properties of distributed systems we have developed two main automation tools. The first one is a rewriting tool that consists in using the Inductive Logical Forms mentioned in Section 1 in order to prove properties by induction on causal order. The second one consists in the automation of standard patterns of reasoning on state machines (such as `QuorumState` discussed in Section 2.3).

3.3.1 Inductive Logical Form

ILFs are declarative logical statements that precisely answer questions such as: "What led the agent at location loc_1 to send a vote to the agent at location loc_2?", in terms of the content of input messages and the states of state machines. ILFs are automatically generated from main classes using various logical simplifications, and mainly using characterizations of the LoE combinators used in the class. For example, one of the most simple but subtle such characterization is the one for the LoE parallel combinator:

$$v \in X \,||\, Y(e) \iff \downarrow(v \in X(e) \vee v \in Y(e))$$

It simply says that v is produced by $X\,||\,Y$ iff it is produced by either one of its two components. As mentioned in footnote 5, the "squash" operator $\downarrow$ is used to enforce proof irrelevance. Informally, in the above formula, the use of $\downarrow$ means that just by knowing that $X\,||\,Y$ produced v, we cannot in general know whether v was produced by X or by Y. For example if two identical replicas run in parallel, and receive the same inputs, then for an external observer there is no way to distinguish between their outputs if they do not label them with different tags.

Given a main class X, we start from an expression of the form $v \in X(e)$, and keep on rewriting that expression using formulas such as the one presented above, to finally obtain a formula of the form $v \in X(e) \iff C$, where C is a complete declarative characterization of the outputs of X. As mentioned above we also apply various logical simplifications to C. Finally, we have built a

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

[5] The $\downarrow$ type operator, called "squash", enforces proof irrelevance, 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.

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

```
∀[Cmd:{T:Type| valueall-type(T)} ]. ∀[clients:bag(Id)]. ∀[cmdeq:EqDecider(Cmd)]. ∀[flrs:ℤ]. ∀[reps:bag(Id)].
∀[f:headers_type{i:l}(Cmd)]. ∀[es:EO+(Message(f))]. ∀[e:E]. ∀[d:ℤ]. ∀[i:Id]. ∀[n,r:ℤ]. ∀[v:Cmd]. ∀[sender:Id].
  (<d, i, make-Msg(‘‘vote‘‘;<<<n, r>, c>, sender>)> ∈ main(Cmd;clients;cmdeq;flrs;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)                4
               ∧ (header(e’) = ‘‘vote‘‘)
               ∧ (n’ = (fst(fst(fst(msgval(e’))))))
               ∧ (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)         8
                       ∧ (header(e1) = ‘‘vote‘‘)
                       ∧ (n’ = (fst(fst(fst(msgval(e1))))))
                       ∧ (r’ = (snd(fst(fst(msgval(e1))))))
                       ∧ (c’’ = (snd(fst(msgval(e1)))))))
                    ∧ (NewRoundsStateFun(Cmd;f;n’;es.e’;e1) < r’)
                    ∧ (e = e1)))))))))
```

proof tactic that automatically proves that this double implication is true.

An ILF provides a characterization of all the (internal and output) messages sent by a system. However, it is often useful to get these characterizations for specific kinds of messages, for example to answer questions such as the one presented at the beginning of this section. Therefore, we generate instances of the ILF for all the kinds of messages that the system outputs. For example, for `SC`, we generate characterizations of the sending of ``*vote*``, ``*retry*``, ``*decided*``, and ``*notify*`` messages.

Figure 3 shows the ILF instance for ``*vote*`` messages as generated by Nuprl. For space reasons, we do not show the entire ILF generated for `SC`. The details of this formula are not critical for understanding the methodology described in this section. However, let us explain, at a high level, how it characterizes the sending of ``*vote*`` messages. This formula says that a vote of the form `<<n,r>,c>,sender>` (where `<_,_>` is Nuprl's pair constructor) is sent by `SC` at event `e` to location `i` (see box 1) iff:

- (box 2): the event `e` happens at one of the replica locations, which we call R,
- (box 3): `i` is also a replica location,
- (box 4): there exists a proposal `<n’,c’>` that was received by R either in a ``*propose*`` message or in a ``*vote*`` message at a prior event `e’`,
- (box 5): the proposal `<n’,r’>` is such that `n’` has never been received by R prior to `e’` (for the purpose of this discussion there is no important distinction between `ReplicaStateFun` and the event class `ReplicaState`, which maintains the list of slot numbers that have been proposed so far),
- (box 6): the proposal `<n’,r’>` is such that no decision has been made about `n’` between the events `e’` and `e`,
- (box 7): finally, either `<n,c>` is `<n’,c’>` and is being voted for at the initial round `r=0` in response to the ``*propose*`` or ``*vote*`` message mentioned above that led to a new `Voting` process being spawned,
- (box 8): or `<n,c>` comes from a ``*retry*`` or a ``*vote*`` message, and `r` is not the initial round, meaning that either some replica believed that consensus could not have been reached at round `r-1` (in the case of a ``*retry*`` message), or R was still working on a smaller round number when it received `r` (in the case of a ``*vote*`` message), and is now voting at round `r`.

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

3.3.2 State Machine Properties

As mentioned above in Section 2.3, the `Memory` and `State` EventML keywords allow one to define Moore machines. 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 kind of local properties of `Memory` and `State` state machines, called: *invariant*, *ordering*, *progress*, and *memory*.

Let us informally present the meaning of each of these properties. A state machine invariant is a unary property that is true for all possible states of the state machine. A state machine ordering property is a binary property that is true about all temporarily ordered pairs of states of the state machine. A state machine progress property w.r.t. some predicate P is a binary property that is true about all temporarily ordered pairs of states of the state machine, 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 future state. Such memory properties are used to specify that state machines keep track in their states of some parts of the inputs they receive.

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 X satisfies a predicate P that is one of these four state machine properties, we simply have to instantiate the corresponding general lemma and prove the simpler transition property.

We have developed an annotation language to state these properties in EventML, and have developed general purpose tactics in Nuprl that try to prove these properties automatically (and often succeed). These tactics try various simplifications and use simple reasoners on lists, integers, etc. Let us now describe and illustrate these four kinds of properties using the `QuorumState` and `NewRoundsState` state machines. The latter is defined in Figure 2 above and keeps track of the current round number for each slot number.

Invariant. Invariant properties of state machines are properties that are true about all states of the state machine. A simple example of an invariant of a `QuorumState` state of the form `(cmds,locs)` is that `locs` does not contain repetitions, and `cmds` and `locs` have equal length. Let us call that invariant `quorum_inv`.

Let T be a type, P be a function of type $T \rightarrow \mathbb{P}$, i.e., a unary property on T, where $\mathbb{P}$ is the proposition type. Now, let `SM` be the state machine (`Memory(init,transition,X)`) of type `Class`(T). Informally, `SM` satisfies the invariant property P, i.e., at any event e the state `state` of `SM` satisfies P—we write $P[\texttt{state}]$[6], if P satisfies the following induction principle: (base-case) the initial state satisfies P; (induction-case) at every event e, if (1) `SM` observes `state`, (2) $P[\texttt{state}]$, and (3) `X` observes `x` then $P[(\texttt{transition loc}(e)\ \texttt{x state})]$.

We state the `quorum_inv` invariant 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 semantic meaning of this EventML declaration is the following Logic of Events formula (where `||_||` is how we display the length function in Nuprl):

```
∀Cmd:Type.
∀eo:EO'. ∀e1:E.
∀cmd_num,round:ℤ.
∀cmds:Cmd List. ∀locs:Id List.
  ((cmds,locs) ∈ (QuorumState(Cmd) <cmd_num,round>)(e)
  ⇒ no_repeats(Id;locs) ∧ (||cmds|| = ||locs||))
```

The Nuprl tactic we have designed tries to automatically prove this statement by unfolding `QuorumState`'s definition to a `Memory` class and by instantiating the corresponding general lemma. It (mainly) remains to prove that the base and induction properties are satisfied.

[6] In general, we write $P[x_1; \ldots; x_n]$ iff $(P\ x_1\ \cdots\ x_n)$

For our simple invariant, we have to prove the two following simple facts:

- base-case: `no_repeats(Id;[])` and `||[]|| = ||[]||`;
- induction-case: according to the definitions of `add_to_quorum` and `addvote` used to define `QuorumState` in Figure 1, we have to prove that (`no_repeats(Id;locs)` and `||cmds|| = ||locs||`) implies that (`no_repeats(Id;sender.locs)` and `||c.cmds|| = ||sender.locs||`), provided that `sender` is not already in `locs` (see definition of `add_to_quorum` and `newvote` in Figure 1).

A trivial but important remark is that 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`.

Let us provide another example regarding `NewRoundsState`. An invariant of a `NewRoundsState` state machine is that its state is a positive integer. We express this invariant in EventML as follows:

```
invariant rounds_pos on round
      in (NewRoundsState n)
  == 0 <= round ;;
```

Ordering. Ordering properties express relations between two states at the same location. For example, if `QuorumState` observes `(cmds1,locs1)` at e_1 and `(cmds2,locs2))` at e_2 such that $e_1 \leq_{\texttt{loc}} e_2$, then `cmds1` and `locs1` are final segments of `cmds2` and `locs2` respectively. We call that ordering property, `quorum_fseg`.

Let T be a type, R be a function of type $T \rightarrow T \rightarrow \mathbb{P}$, i.e., a binary relation on T. Let `SM` be the state machine (`Memory(init,transition,X)`) of type `Class`(T). Informally, `SM` satisfies the ordering property R if for all events e_1 and e_2 such that $e_1 \leq_{\texttt{loc}} e_2$, and for all `state1` and `state2` observed by `SM` at e_1 and e_2 respectively, then $R[\texttt{state1}; \texttt{state1}]$ provided that R is a reflexive and transitive relation and every two consecutive states are in relation w.r.t. R.

We state `quorum_fseg` in EventML as follows, where fseg is a predicate available in Nuprl's library:

```
import fseg
ordering quorum_fseg
      on (cmds1,locs1) then (cmds2,locs2)
      in QuorumState ni
  == fseg ::Cmd cmds1 cmds2
  /\ fseg ::Loc locs1 locs2 ;;
```

Let us provide another example, regarding `NewRoundsState`. An ordering progress property of `NewRoundsState` states is that rounds can only increase over time. We express this property in EventML as follows:

```
ordering rounds_inc on round1 then round2
      in (NewRoundsState n)
  == round1 <= round2 ;;
```

Progress. Note that at any event, a `NewRoundsState` state can increase only if `RoundInfo` observes a vote or a retry for a round number strictly greater than the state. We call such a property *progress*, because some progress has actually been made, i.e., the state has changed.

Let T and U be types, R be a function of type $T \rightarrow T \rightarrow \mathbb{P}$, i.e., a binary relation on T, and P be a function of type $U \rightarrow T \rightarrow \mathbb{P}$. Let `X` be a class of type `Class`(U), and `SM` be the state machine (`Memory(init,transition,X)`) of type `Class`(T). Informally, `SM` satisfies the progress property R if for all `state1` and `state2` of types T, $R[\texttt{state1}; \texttt{state2}]$ is true provided that there exists two events e_1 and e_2 such that the following holds:

- $e_1 <_{\texttt{loc}} e_2$,

- R is a transitive relation and P is decidable,
- `state1` and `state2` are observations made by `SM` at e_1 and e_2 respectively,
- R satisfies the following property: for all events e such that `X` observes `x` at e and `SM` observes `state` at e, if P[`x`; `state`] then R[`state`; `transition x state`], otherwise `state` has to be equal to `transition x state` (i.e., progress can only be made when P is satisfied), and
- there exists an event e such that $e_1 \leq_{\mathtt{loc}} e$, $e <_{\mathtt{loc}} e_2$, `SM` observes `state` at e, `X` observes `x` at e, and P[`x`; `state`] (i.e., progress has been made between e_1 and e_2).

For example, if `(NewRoundsState n)` observes `round1` at event e_1 and `round2` at a latter event e_2, such that progress has been made between e_1 and e_2, then `round1 < round2`, where our progress property is the function:

$$\lambda((n', r'), cmd).\lambda r.(n = n' \ \wedge \ r < r')$$

We state this property in EventML as follows:

```
progress rounds_strict_inc
      on round1 then round2
      in (NewRoundsState n)
  with ((n',round'),cmd) in RoundInfo
   and round => n' = n /\ round < round'
  == round1 < round2;;
```

Memory. Memory properties express that state machines do not ignore inputs. For example, if `(NewRoundsState n)` observes `round1` at event e_1 and `round2` at a strictly latter event e_2, `RoundInfo` observes `((n',r'),cmd)` at e_1, and `n = n'` then `r' <= round2`. We call that property, `rounds_mem`.

Let T and U be types, and R be a function of type $U \to T \to T \to \mathbb{P}$. Let `X` be a class of type `Class`(U), and `SM` be the state machine `(Memory(init,transition,X))` of type `Class`(T). Informally, `SM` satisfies the memory property R if for all events e_1 and e_2 such that $e_1 <_{\mathtt{loc}} e_2$, for all `x` observed by `X` at e_1, and for all `state1` and `state2` observed by `SM` at e_1 and e_2 respectively, then R[`x`; `state1`; `state2`] is true provided that R satisfies the following property: for all events e and e' such that $e_1 \leq_{\mathtt{loc}} e$, $e <_{\mathtt{loc}} e'$, and $e' <_{\mathtt{loc}} e_2$, for all `x1` and `x2` observed by `X` at e and e' respectively, for all `state1` and `state2` observed by `SM` at e and e' respectively, if R[`x1`; `state1`; `state2`] then R[`x1`; `state1`; `transition x2 state2`].

We state `rounds_mem` in EventML as follows:

```
memory rounds_mem
    on round1 then round2
    in (NewRoundsState n)
  with ((n',round'),cmd) in RoundInfo
  == (n = n') => round' <= round2
;;
```

3.4 Proof of Agreement

To prove agreement we first prove several simple lemmas.

(1) In any round, each instance of `Replica` votes for at most one command: This follows from the fact that a replica votes at most once per round.

(2) Two ``*notify*`` messages sent in the same round must be for the same command: If at most $3 * \mathtt{flrs} + 1$ votes can be cast, and two different `Quorum` classes receive $2 * \mathtt{flrs} + 1$ unanimous votes from distinct voters, then both of those unanimous votes must be for the same command.

(3) If a ``*notify*`` message for c and a ``*retry*`` message for d are sent in the same round, then $c = d$: This is another counting argument. If $2{*}\mathtt{flrs}{+}1$ votes have been cast for c, then the majority of the votes in *any* collection of $2 * \mathtt{flrs} + 1$ votes (in that round) must be for c; so every ``*retry*`` will be for c.

(4) A vote for command d at round (n, j) such that $j > 0$, can always be traced back to a ``*retry*`` for d at round $(n, j - 1)$. (This is proven by induction on the well-founded causal ordering of events.)

Lemma (2) leaves us with the interesting case: Suppose that in round (n, i) some instance of `Replica` detects a consensus for proposal (n, c). We must show that if $k > i$, then a consensus detected in round (n, k) must also be for (n, c).

We prove that by showing something stronger: If $k > i$, then *every* vote in round (n, k) will be for (n, c). We prove this result by induction on $k - i$. If $k - i = 1$, it follows from lemma (4) by appealing to lemma (3). Otherwise, it follows from lemma (4) by appealing to the induction hypothesis on $k - 1$.

3.5 Proof of Validity

Validity is a corollary of the following lemma, which we prove by induction on the causal order of events:

$$\begin{array}{l}\forall e\colon \mathtt{E}.\ \forall n\colon \mathbb{Z}.\ \forall c\colon \mathtt{Cmd}.\ \forall r\colon \mathbb{Z}.\ \forall loc\colon \mathtt{Loc}.\\ \left(\begin{array}{l} \quad (n, c) \in \mathtt{decided'base}(e) \\ \vee\ (((n, r), c), loc) \in \mathtt{vote'base}(e) \\ \vee\ ((n, r), c) \in \mathtt{retry'base}(e) \end{array}\right)\\ \quad \Rightarrow\ \downarrow\exists e'\colon \mathtt{E}.\ e' < e\ \wedge\ (n, c) \in \mathtt{propose'base}(e')\end{array}$$

3.6 Proof Effort

Thanks to the rich library of definitions, facts, and proof tactics about LoE and GPM that we developed over the past several years, and also thanks to our new automation tools, we have specified 2/3 consensus and have completed in Nuprl the proofs of its two safety properties 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 Section 1, the semantic meaning of an EventML program can be expressed by a LoE event class and by a GPM program. We carry out our correctness proofs on the LoE description of the main event class. In order to trust the program we run, we prove that it implements that description, i.e., that it outputs exactly the same observations.

The GPM program generated by EventML is a collection of processes, represented as a function from locations to processes. In order to generate a finite system of distributed processes, we need to instantiate this function with a finite number of locations. These locations can be obtained from the EventML specification. For example, in Section 2.1, `SC` is declared as the `Replica` process running at each location in `reps`.

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 combinators PC such that we have proved that PC implements LC.

Consider the parallel combinator. Let X_1 and X_2 be event classes of type T that are implementable by pr_1 and pr_2, respectively. The LoE parallel combinator $X_1 \,||\, X_2$ is defined as follows:

$$\lambda eo.\lambda e.(X_1\ eo\ e) + (X_2\ eo\ e)$$

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 for the LoE and GPM parallel combinators):

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

This expression defines a function that takes a location loc and returns a process that runs p_1 and p_2 in parallel at that location. This process is defined as a co-recursive program that maintains a state composed of two processes: its two components. Its initial state is $(pr_1\ loc, pr_2\ loc)$. If the current state of the process is the pair (p_1, p_2), then if both p_1 and p_2 have already halted, i.e., they are the special process `halt`, then the process becomes the halted process `halt`. Otherwise, the process waits for an input message, and once it has received such a message, say m,

(1) for $i \in \{1, 2\}$, it applies p_i to m to obtain a new process p'_i and a bag of output values out_i[7],

(2) it combines these bags using the append operation on bag (+) to form the bag of output values out, and finally,

(3) it outputs out and co-recursively calls itself on the new state (p'_1, p'_2).

The complexity of that expression and of the proof that it implements $X_1\,||\,X_2$ comes from the fact that processes can either be of the form `halt` or of the form $\texttt{run}(f)$. Because of the similar structure in the two expressions, it is nonetheless easy to prove that $pr_1\,||\,pr_2$ implements $X_1\,||\,X_2$. The same is true for the other combinators.

5. Related Work

IOA. IOA [16, 17, 20, 39] 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 [39]. Both I/O automata and event classes can specify I/O observations of distributed systems. A major difference between IOA and the Logic of Events is that IOA is state-based, while the Logic of Events is event-based (states are implicitly maintained by recursive combinators). Our respective approaches differ in the sense that the programming/logical dualism of ours allows us to both prove protocol properties and generate code within the Nuprl proof/programming environment, 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" [28]. TLA$^+$ [13, 28, 29] is a language for specifying systems described in TLA. TLAPS "is a platform for the development and mechanical verification of TLA$^+$ proofs" [13]. 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 to prove the correctness of a protocol. 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 (note that we have not yet proved such properties either). TLA$^+$ does not perform program synthesis.

Orc. Orc [22, 23] is a programming language for structured concurrent programming. It is based on a small set of combinators to "orchestrate the concurrent invocations of sites" [22] that perform basic services (such as timers). Expressions in Orc are similar to our event classes. Among other things, Orc has similar parallel and delegation combinators and allows recursive definitions. Although, many formal semantics of Orc have been defined, to the best of our knowledge none of them are formalized in a theorem prover.

FVN. A similar approach to ours is the one taken by the FVN framework [41]; their system and ours have the same general structure. They use the NDlog declarative networking language as the bridge between high-level logical specifications and low level programs. NDlog corresponds to EventML in our framework. NDlog programs can be translated to logical statements expressed in PVS [40]. This would correspond to the Logic of Events part of our framework. Using P2 [31], NDlog programs can also be compiled to dataflow programs. This would correspond to the level of our General Process Model.

sel4. Our approach is also similar to the one taken by Klein et al. to verify the sel4 microkernel [24]. In their methodology, they use Haskell as their specification language. This roughly corresponds to the level of abstraction of EventML in our framework, even though they can run the Haskell code, while in our framework executing EventML requires extracting its semantic meaning in terms of Nuprl GPM processes. Then, they translate this code to an Isabelle/HOL version. They prove that this "executable specification" refines an "abstract specification", which corresponds to the level of the Logic of Events. 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 the level of our General Process Model. Among other things, our paper shows that a similar methodology can be used to design and implement correct fault-tolerant distributed systems.

6. Conclusion and Future Work

Our methodology scales to more complicated and subtle distributed protocols. For examples, we have written several EventML specifications of the Multi-Paxos protocol [27, 35] and proved its safety properties to be correct in Nuprl. We are also building a generic correct-by-construction ordered broadcast service that can switch between various consensus protocols such as 2/3-consensus and Paxos Synod.

In order to get efficient code, we are building, within Nuprl, a formal tool that we have tuned to automatically optimize GPM programs and prove that the optimized code and the non-optimized program are bisimilar [34], using new untyped reasoning techniques. We are also experimenting with a program translator to Lisp. Using this translator, we obtain code that performs decently.

We are also building support in both 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.

References

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

[2] *22nd ACM Symp. on Operating Systems Principles*. ACM, 2009.

[3] *37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM, 2010.

[7] The application of a process p to a message m is defined as follows: if $\texttt{halted}(p)$ then return $(\texttt{halt}, \{\})$, otherwise p is of the form $\texttt{run}(f)$, and therefore, return $(f\ m)$ that computes to a pair process/bag of outputs.

[4] 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.

[5] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[6] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, and James J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *22nd IEEE Computer Security Foundations Symp.*, pages 124–140. IEEE Computer Society, 2009.

[7] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* [3], pages 445–456.

[8] 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.

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

[10] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[11] Mark Bickford, Robert L. Constable, Joseph Y. Halpern, and Sabina Petride. Knowledge-based synthesis of distributed systems using event structures. *Logical Methods in Computer Science*, 7(2), 2011.

[12] 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.

[13] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying Safety Properties With the TLA+ Proof System. In Jürgen Giesl and Reiner Haehnle, editors, *Fifth International Joint Conference on Automated Reasoning - IJCAR 2010*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 142–148, Edinburgh, United Kingdom, 2010. Springer.

[14] 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.

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

[16] 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.

[17] 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.

[18] Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session types for inter-process communication. Technical Report TR-2003-133, University of Glasgow, 2003.

[19] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* [3], pages 299–312.

[20] 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.

[21] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[22] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR 2006 - Concurrency Theory, 17th Int'l Conf.*, volume 4137 of *LNCS*, pages 477–491. Springer, 2006.

[23] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In *Formal Techniques for Distributed Systems*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[24] 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 *22nd ACM Symp. on Operating Systems Principles* [2], pages 207–220.

[25] Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 2002. `www.nuprl.org/html/02cucs-NuprlManual.pdf`.

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

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

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

[29] Leslie Lamport. *TLA^{+2}: A Preliminary Guide*, 2011. `msr-inria.inria.fr/~doligez/tlaps/`.

[30] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating Systems Principles* [2], pages 321–334.

[31] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *SOSP*, pages 75–90. ACM, 2005.

[32] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.

[33] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symp. on Security and Privacy*, pages 165–179. IEEE Computer Society, 2011.

[34] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in Nuprl using computational equivalence and partial types. Accepted for publication to ITP, 2013.

[35] Robbert Van Renesse. Paxos made moderately complex. `www.cs.cornell.edu/courses/CS7412/2011sp/paxos.pdf`, 2011.

[36] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[37] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *Workshop on Future of Software Engineering Research*, pages 343–348. ACM, 2010.

[38] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 266–278. ACM, 2011.

[39] Joshua A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Departement of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[40] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 61–75, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. Formally verifiable networking. In *HotNets*. ACM SIGCOMM, 2009.