EventML: Specification, Verification, and Implementation of Crash-Tolerant State Machine Replication Systems

Vincent Rahli[☆]

SnT, University of Luxembourg, Luxembourg

David Guaspari

Mark Bickford

Cornell University, Ithaca, NY, USA

Robert L. Constable

Cornell University, Ithaca, NY, USA

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 logical foundations as well as 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: Nuprl; EventML; Functional programming; Formal methods; Formal verification; Interactive theorem proving; Distributed systems; Fault tolerance; Event logic; Event-based programming

[☆]This work was partially supported by the DARPA CRASH project, award number FA8750-10-2-0238, by the SnT, and by the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.

Email address: vincent.rahli@gmail.com (Vincent Rahli[☆])

URL: markb@cs.cornell.edu (Mark Bickford), rc@cs.cornell.edu (Robert L. Constable)

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.

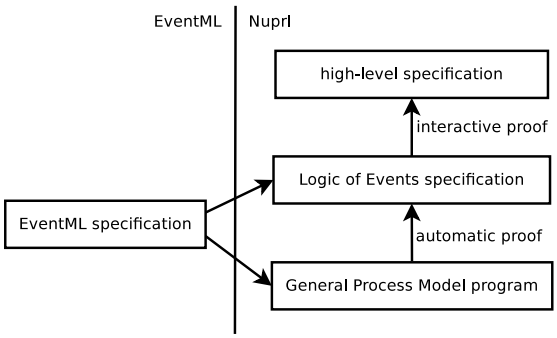
One reason that distributed systems are especially difficult to code correctly and maintain is that there are many intricate failure scenarios to consider. Failure scenarios can be hard to generate and testing them all is usually not possible. Model checkers are often used to verify that distributed systems are correct [64, 3, 14, 30, 59]. 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 allow one to provide definitive arguments.

We use the EventML language to develop protocols. EventML works synergistically with the Nuprl proof assistant [21, 4], which is closely related to the Coq [9, 22] proof assistant. Nuprl is a programing/logical environment based on Constructive Type Theory (CTT) [21, 4], that allows one both to prove mathematical results, and to program and prove properties about these programs, and do this in a single formal method tool.

EventML. EventML is a domain-specific ML-like functional programming language for distributed protocols based on asynchronous message passing. It allows developing 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) [10, 11] to specify and reason about the information flow of distributed program runs, (2) a General Process Model (GPM) [12] 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 is one of two things: the "possibly" still running process $\texttt{run}(f)$ where f is a function that given an input of type A, generates a (possibly empty) bag[1] of outputs of type B, and becomes a possibly different process; or a special value called `halt`, that is used to denote a terminated process. Formally, a process that takes inputs of type A, and outputs elements of type B, is an element of the following coinductive type (the definition of the Nuprl `corec` type is outside the scope of this paper): $\texttt{corec}(\lambda P.(A \rightarrow P \times \texttt{Bag}(B))+\texttt{Unit})$, where `Unit` is a singleton type; $\times$ is the type of pairs; and $+$ is the disjoint union type, i.e., the type of left and right injections. Therefore $\texttt{run}(f)$ is defined as $\texttt{inl}(f)$, and `halt` is defined as $\texttt{inr}(\star)$, where $\star$ is `Unit`'s only inhabitant. 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) [10, 11], related to Lamport's notion of causal order [39], 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 [60] and cyber-physical systems [5]. In the context of this paper, an event is an abstract entity corresponding to the reception of a message[2]. 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 specify a system by describing how it reacts to events.

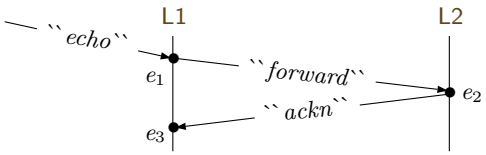
Automation. Formally verifying distributed protocols is not trivial and can be time consuming. However, because we are using a tactic-based proof assistant in the style of LCF [28], there is much room for automation. We have built two main automation tools to assist us in proving properties of distributed systems.

From an EventML specification we automatically generate an *inductive logical form* (ILF), a first-order formula that characterizes the response to any

[1] In Nuprl, the bag type is the list type quotiented by the permutation relation.

[2] Formally events are more general than that because they might correspond to something else than just the reception of messages.

Figure 1 Example of a message sequence diagram



event e in terms of the information computed at locally prior events (hence the word *inductive*). 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 specifications are typically composed of several small state machines.

Contributions. This paper introduces basic ideas of EventML, which is a domain specific programming language (DSL) that implements a paradigm in which programmers can flexibly use constructive proof assistants such as Nuprl or Coq to develop verified distributed programs. EventML itself is implemented in SML [47]. One characteristic of EventML is that it relies on combinators that have well-defined semantics in both GPM and LoE, which allows us to automate a large part of the reasoning necessary to connect the correctness of the running code to high-level correctness arguments. First, we introduce our Logic of Events in Sec. 2. Then, we show how EventML can be used to define a non-trivial fault-tolerant consensus protocol in Sec. 3, prove the safety properties of this protocol in Sec. 4 using automation tools described in Sec. 5, and generate a verified implementation in Sec. 6, all of that in a single formal method tool. Sec. 7 presents EventML's syntax and static semantics. 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 [39, 57]. More material can be found at the following address: `http://nuprl.org/KB/show.php?ID=709`. This paper is based on results presented in [53, 55].

2. The Logic of Events

2.1. Event Orderings

In order to reason about a distributed system, one often reasons about its possible runs, which are sometimes modeled as execution traces [58]. In LoE, system runs are captured using the notion of an *event ordering*. 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 (Fig. 1 provides an example of a simple message sequence diagram). As opposed to [58], a trace here is not just one sequence of events but instead can be seen as a collection of local traces (one local trace for each process),

where a local trace is a collection of events all happening at the same location and ordered in time, and such that some events of different local traces are causally ordered. For example, when interpreted as a trace, Fig. 1 shows two local traces, one at location L1 and one at location L2. We express system properties as predicates on event orderings. A system satisfies such a property if every possible execution of the system satisfies the predicate.

An event ordering is formally defined as a dependent record, which can be expressed as follows using some ML-like notation:

```
type EventOrdering = {
  Event       : Type;
  loc         : Event → Loc;
  causalOrder : Event → Event → Prop;
  pred        : Event → Event;
  info        : Event → Message
}
```

where Loc is the type of physical locations, which could for example be the type of pairs of the form IP address/port number; loc is a function that associates a physical location with each event; causalOrder is a *causal ordering* relation on events [39], which we write as $<$; pred is a function that given an event e, either returns its local predecessor if it has one, or returns e if it does not have one; info associates primitive information with each event, which, in this paper, is simply a message.

A message is a pair of a header h and a body b, such that the type of b depends on h. To achieve this, each specification provides a function from headers to types. In this paper we represent a header as a list of characters surrounded by double back-quotes. Given a message m of the form (h, b), $\mathbf{header}(m)$ returns h, and $\mathbf{body}(m)$ returns b.

We define $e <_{\mathtt{loc}} e'$ as $e < e' \wedge \mathtt{loc}(e) = \mathtt{loc}(e')$. We sometimes write EO for the type of event orderings. For readability our notation often suppresses event ordering variables. Thus we write Event for the component of event ordering eo that specifies its events, rather than writing Event(eo)—and we do the same for the other components. We also sometimes write E instead of Event.

The components of an event ordering must satisfy the following axioms:

- The causal ordering relation $<$ is a transitive and well-founded relation.
- Equality between events is decidable.
- The local predecessor of an event e happens at the same location as e: loc(e)=loc(pred(e)).
- If an event e is not the initial event at its location, i.e., pred(e)=e' such that $e \neq e'$, then e' happens causally before e, i.e., $e' < e$.
- An event e is the initial event at its location, i.e., pred(e)=e iff for all event e' such that loc(e)=loc(e'), $e < e'$.

- The predecessor function is injective, i.e., if $\mathsf{pred}(e)=\mathsf{pred}(e')$ then $e = e'$.
- If $e_1 <_{\mathtt{loc}} e_2$ and $\mathsf{pred}(e_2)=e$ then either $e = e_1$ or $e_1 < e$.

For example, the message sequence diagram presented in Fig. 1 depicts a simple event ordering. Event e_1 corresponds to the reception 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, i.e., $e_1 < e_2$, $e_2 < e_3$, and $e_1 <_{\mathtt{loc}} e_3$.

2.2. Event Observers

In LoE, we specify systems by defining and combining *event observers* [10]. An event observer is an abstraction of a process. It is a function that assigns to any event ordering *eo* and event e in the event ordering *eo*, an unordered bag of outputs observed/produced at e. The type of event observers that observe expressions of type T is formally defined as follows:

$$\mathtt{Obs}(T) = eo{:}\mathtt{EO} \rightarrow e{:}\mathtt{E} \rightarrow \mathtt{Bag}(T)$$

For example, the following observer of type $\mathtt{Obs}(\mathtt{Loc})$, where Loc is the type of locations, recognizes every event and observes its location: $\lambda eo.\lambda e.\{\mathtt{loc}(e)\}$, where $\{v\}$ is the singleton bag containing v.

Event observers can 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 above location observer, recognizes all events and handles them by returning their locations. Sec. 3 below introduces several event observers. 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[3]. Formally, a base observer that recognizes headers of the form h is defined as follows:

$$\lambda eo.\lambda e.\mathtt{if}\ h{=}\mathtt{header}(\mathtt{info}(e))\ \mathtt{then}\ \{\mathtt{body}(\mathtt{info}(e))\}\ \mathtt{else}\ \{\}$$

where $\{\}$ is the empty bag. 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. Note that an event

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

observer can make observations at several locations, possibly at an infinite number of locations if the location type is infinite. This is for example the case of the location observer defined above, which observes the location of every event.

2.3. Event Observer Relation

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 $(X\ eo\ e)$. As mentioned above, 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 expressions, such as messages, 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".

2.4. Event Observer Characterization

To reason about a system, one reasons about the observations of its main observer. Typically, a safety property of a system specified by a main event observer X is of the form: if $(l_1, m_1) \in X(e_1)$, ..., $(l_n, m_n) \in X(e_n)$, then some property P holds about m_1, ..., m_n. Again, typically, one proves such a property by tracing back the outputs (l_1, m_1),...,(l_n, m_m) to inputs of the system, from which P follows. Therefore, to trace back outputs to inputs, we have to be able to characterize the observations of an event observer in terms of the observations made by its sub-components.

Parallel Combinator. Let us illustrate how this is done using our parallel combinator $X\,||\,Y$, which, in terms of processes, runs the two processes X and Y in parallel. This combinator is formally defined as follows:

$$\lambda eo.\lambda e.(X\ eo\ e)\ \texttt{+}\ (Y\ eo\ e)$$

i.e., it observes the outputs of both X and Y at e. The operator `+` is the append operator on bags, i.e., of type $\texttt{Bag}(T) \to \texttt{Bag}(T) \to \texttt{Bag}(T)$, and this for any type T. The observations made by our parallel operator are characterized as follows:

$$v \in (X\,||\,Y)(e) \iff \downarrow(v \in X(e)\ \vee\ v \in Y(e)) \tag{1}$$

This says that v is produced by $X\,||\,Y$ iff it is produced by either of its components. The squashing operator $\downarrow$ enforces "proof erasure". Intuitively, it is needed because just by knowing that $X\,||\,Y$ 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. More

precisely, because Nuprl implements a constructive logic, disjunctions are inhabited by left and right injections, which are tags that tell us whether we have a proof of either the left or the right disjunct. Therefore, if we had a proof of $v \in X(e) \ \vee \ v \in Y(e)$ then this proof would either be an injection left and we would know that v comes from X; or an injection right and we would know that v comes from Y. Intuitively, proof erasure says that, when deducing any consequence of $v \in (X \,||\, Y)(e)$ we are not allowed to assume that we know which.

Delegation Combinator. Let us now present another crucial event observer. The delegation (or spawning) combinator $X \gg= Y$ is used to spawn-off and delegate tasks to sub-processes as specified by Y. This combinator is especially useful for compositional reasoning. It is formally defined as follows:

$$X \gg= Y = \lambda eo.\lambda e.\texttt{beforeEq}(e) \gg=_b (\lambda e'.X\ eo\ e' \gg=_b (\lambda x.Y\ x\ (eo\texttt{.}e')\ e))$$

where $\texttt{beforeEq}(e)$ is the causally ordered list of events that are local predecessors of or equal to e, which can easily be computed using pred; given a bag B of type $\texttt{Bag}(T)$ and a function F of type $T \to \texttt{Bag}(U)$, $B \gg=_b F$ is the bind operator of the bag monad [48], which is defined as $\texttt{bunion}(\texttt{bmap}(F, B))$ returning a bag of type $\texttt{Bag}(U)$, where bunion is the union operation on bags, i.e., of type $\texttt{Bag}(\texttt{Bag}(T)) \to \texttt{Bag}(T)$, and this for any type T, and bmap is the map operation on bags, i.e., of type $(T \to U) \to \texttt{Bag}(T) \to \texttt{Bag}(U)$, and this for any type T; and $eo\texttt{.}e$ is the event ordering restricted to the events not happening at location $\texttt{loc}(e)$ or happening at location $\texttt{loc}(e)$ but not prior to e, i.e., $eo\texttt{.}e$ is the event ordering containing all the events of eo except those happening locally before e.

The delegation observer uses observations made by X to start sub-observers as described by Y: for each observation x made by X at some event e, we start the new observer $Y(x)$ at event e. An observation v made by $X \gg= Y$ at some event e is then an observation made by a $Y(x)$ at e such that x was observed by X at some locally earlier event e', which is when the observer $Y(x)$ started, i.e., assuming $X \in \texttt{Obs}(T)$:

$$v \in X \gg= Y(e) \iff \downarrow\exists e' <_{\texttt{loc}} e.\ \exists x : T.\ x \in X(e') \ \wedge \ v \in Y(x)(e) \qquad (2)$$

Again, the squashing operator $\downarrow$ is required because in general we cannot pinpoint the event at which $Y(x)$ started.

Event Observer Monad. We use the symbol $\gg=$ because the event observer type forms a monad having $\gg=$ as its bind operator, which comes from the fact that $\gg=$'s definition relies on the bind operator of the bag monad. The return operator of the event observer monad is:

$$\texttt{return}(v) = \lambda eo.\lambda e.\texttt{if first}(e)\ \texttt{then}\ \{v\}\ \texttt{else}\ \{\}$$

where $\texttt{first}(e)$ is true iff e has no local predecessor, i.e., $\text{pred}(e)=e$. The event observer monad satisfies the left identify law

$$\texttt{return}(v) \gg= Y = Y(v)$$

the right identity law

$$X \texttt{ >>= } \lambda v.\texttt{return}(v) = X$$

and the associativity law

$$(X \texttt{ >>= } Y) \texttt{ >>= } Z = X \texttt{ >>= } (\lambda v.(Y(v) \texttt{ >>= } Z))$$

If we had not used `first` in `return`'s definition, then the left identity law for example would not be true, because $\texttt{return}(v)$ `>>=` Y would spawn-off a new sub-process $Y(v)$ at each event.

Parallel as Delegation. Because the delegation combinator can in fact run several sub-processes in parallel, we can define the parallel combinator in terms of the delegation combinator as follows:

$$X \texttt{ || } Y = (\texttt{returnBag}(\{\texttt{tt}, \texttt{ff}\}) \texttt{ >>= } \lambda b.\texttt{if } b \texttt{ then } X \texttt{ else } Y)$$

Where `tt` and `ff` are the true and false Booleans, and $\texttt{returnBag}(b)$ is the following generalization of the `return` combinator: $\lambda eo.\lambda e.\texttt{if first}(e) \texttt{ then } b \texttt{ else } \{\}$.

2.5. Constraining Event Orderings

As mentioned above, we express system properties as predicates on event orderings, and a system satisfies such a property if every possible execution of the system satisfies the predicate. There are not many useful properties that are true about all event orderings. Therefore, to prove something interesting, we have to restrict event orderings to the ones that are possible runs of a given system specification, i.e., of a given main event observer. We do this by defining predicates on the causal ordering relation of event orderings that express the possible communications between the agents of a system. Such predicates highly depend on the system model that one wants to target. By default, LoE is open to a wide range of system models, and one commits to a specific system model using predicates on event orderings. For example, one can assume that messages are never forged by stating that for any main event observer X and event ordering eo, if e is an X-event then there is a prior event e' such that $\texttt{info}(e)$ was produced by X at e', i.e., the process running at $\texttt{loc}(e')$ followed the protocol as described by X. By default, we assume that messages are delivered asynchronously, and may be delivered more than once. To assume a synchronous system, one would have to add a time field in the definition of the event ordering type, that associates time to events as done by Anand and Knepper [5].

3. A Specification of 2/3 Consensus

This section shows how EventML, which is the DSL we have built on top of LoE, can be used to define a non-trivial fault-tolerant consensus protocol, namely the 2/3 consensus protocol [17]. Fig. 2 and Fig. 3 provide the full

Figure 2 2/3 consensus—part 1/2

```
specification two_thirds

(* ========== Parameters ========== *)
(* Command type with equality decider cmdeq *)
parameter Cmd, cmdeq : Type * Cmd Deq
(* max number of failures                   *)
parameter F          : Int
(* locations of (3 * F + 1) replicas        *)
parameter reps       : Loc Bag
(* locations of the clients to be notified  *)
parameter clients    : Loc Bag

(* ========== 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)) ;;

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

Figure 3 2/3 consensus—part 2/2

```
(* ========== 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 RcvProposal =
  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, RcvProposal) ;;
observer NewVoters =
  out_proposal o (RcvProposal, ReplicaState) ;;

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

EventML specification, and Sec.3.1 provides a top-down description of this specification. (Note that the reader does not necessarily need to read the code presented in these figures now. They are displayed here so that the reader can see what the full specification looks like and how the different pieces of code presented below fit together.) The problem we consider is as follows: A system has been replicated for fault tolerance [62]. 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. Therefore, it follows from the FLP impossibility result [23] that 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 F failures, another parameter of the specification, by using $3 * \mathsf{F} + 1$ replicas. (An appealing feature of the protocol is that with a small change, and using $5 * \mathsf{F} + 1$ replicas, it can tolerate Byzantine failures.) The parameter reps is a bag[4] that denotes the locations at which the replicas will execute. We will see below that when proving properties about our specification, we assume that reps has size $3 * \mathsf{F} + 1$. 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 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. 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 [39, 57].

3.1. 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. 3.2 describes one of those state machines, which performs the key computation used to detect consensus. Sec. 3.3 shows how EventML defines an event observer specifying that state machine.

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

[4] We could as well have used a list but we use a bag instead because replicas are not ordered.

in their successive votes, and how 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 (the interface of the protocol is defined in Fig. 2):

```
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 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, i.e., the processes of the system do not react to these events. However, it implicitly declares the functions notify'send and notify'bcast for creating directed messages. In EventML, a directed message has primitive type Interface . Internally, they are represented as pairs of a location and a message. If m is the ``*notify*`` message with body p, then the expression (notify'send l p) is a directed message, which is represented internally by the pair (l, m) instructing that m be sent to l; and the expression (notify'bcast {l1,l2,...}p) is a bag of directed messages, which is represented internally as the bag $\{(\mathsf{l1}, m), (\mathsf{l2}, m), \ldots\}$ of such instructions.

Typically, the complete interface of a system is defined in terms of its input, output, and internal messages. The internal ones are those that are only meant to 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 define the event observer Replica. The main program, which executes the protocol, is specified by the main event observer SC (see Fig. 3):

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

A main observer has type (Interface Obs). As mentioned above, the reps parameter denotes the locations at which the replicas will execute. We may think of the SC observer 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. In LoE, (X @ locs) is defined as:

$$\lambda eo.\lambda e.\texttt{if loc}(e) \in \text{locs } \texttt{then } X\ eo\ e\ \texttt{else } \{\}$$

For each natural number 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. As mentioned above in Sec. 2.4, the spawning/delegation operator "_>>=_" is a primitive which is used by processes to start sub-processes (see Fig. 3):

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

The observer NewVoters decides when to spawn a new voting process. Voter is a higher-order function; the values it returns are observers that do the voting. When some NewVoters-event e occurs and $v \in \text{NewVoters}(e)$, 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* at and after e. For any event e there will be at most one v such that $v \in \text{NewVoters}(e)$. 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 event 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, which is defined as follows (see Fig. 3):

```
observer NewVoters = out_proposal o (RcvProposal, ReplicaState) ;;
```

is a Mealy state machine. It reacts to RcvProposal-events, i.e., to ``*proposal*`` (coming from outside the system) and ``*vote*`` messages (from inside), and it filters those events. At any location *loc*, thanks to the ReplicaState Moore state machine, 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 $2 * F + 1$ votes for (n,c) from $2 * F + 1$ different replicas.

We cannot guarantee that any particular poll of the Voter observers will achieve such a result. Accordingly, for each slot number 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 slot number/round number pairs of the form (n,r). *Ballots* are voting round/command pairs of the form ((n,r),c). Thus, a Voter casts votes for a particular proposal *in a particular round*. *Votes* are ballot/location pairs of the form (((n,r),c),loc). 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 (see Fig. 3):

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

where, as mentioned above in Sec. 2.4, "_||_" 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 (and is garbage collected). We use this to terminate any voting for n once consensus has been reached on n. In LoE (A until B) is defined as:

$$\lambda eo.\lambda e.\texttt{if existsLast}(eo, e, \mathsf{B}) \texttt{ then } \{\} \texttt{ else } \mathsf{A}\ eo\ e$$

where $\texttt{existsLast}(eo, e, \mathsf{B})$ is true iff there exists a B-event in $\texttt{before}(e)$, where $\texttt{before}(e)$ is the causally ordered list of events that are local and strict predecessors of e. Because (A until B) does not execute B once a B-event happens, we use the idiom (A until B) || B, to run A, stop A once a B-event happens, and also run B.

Note that Round (defined in Fig. 2), Rounds, NewRounds, and Notify (defined in Fig. 3) are also functions that return event observers. Let us discuss these in further detail here.

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. The Round observer, detailed further in Sec. 3.2, 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.

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

Finally, Notify(n) handles internal ``*decided*`` messages 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.

3.2. Detecting Consensus

Round ((n,r),c) has two components (see Fig. 2):

```
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. Output(G) is defined in LoE as

$$\lambda eo.\lambda e.\mathsf{G}(\mathtt{loc}(e))$$

The second component 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. Once(A) is defined as (A until A). 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 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 $2 * \mathsf{F} + 1$ 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 $2 * \mathsf{F} + 1$ votes on this round. (Note that if F failures have occurred, no more votes from distinct replicas 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 $2*\mathsf{F}+1$ 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. 4 shows that, for any n, all notifications about the n^{th} command will agree on which command has been chosen.

3.3. Implementing Quorum

Quorum(n,r) is a Mealy machine: in response to inputs it may change state and produce outputs. Let us factor its definition. We first define the Moore machine QuorumState(n,r), 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 appropriate directed messages. EventML provides primitives such as Memory for defining Moore machines (see Fig 2):

```
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 ([c1;c2;...],[l1;l2;...]) means that, in round (n,r), the state machine has thus far received a vote from l1 for c1, a vote from l2 for c2, 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. In this case, the initial state of a replica is independent from its location. (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. This composition combinator is defined in LoE as follows:

$$\lambda eo.\lambda e.\mathsf{X1}\ eo\ e \gg\!=_b (\lambda x_1 \ldots \mathsf{Xn}\ eo\ e \gg\!=_b (\lambda x_n.\mathsf{f}\ \mathtt{loc}(e)\ x_1\ \cdots\ x_n))$$

Quorum is defined as follows:

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

This observer 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 ``*vote*`` events, but not all of them because 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 discards this input by not outputting 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 . is the cons operation on lists, is the command list that results from processing the input.

We can now understand the outer conditional: If its condition is false then we have not seen $2*F+1$ 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 itself, 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.

This concludes the presentation of our EventML specification of the 2/3 consensus protocol. We will now discuss its safety properties, which we have proved in Nuprl.

4. The Safety Properties of 2/3 Consensus

From SC, our EventML specification of 2/3 consensus, EventML's compiler generates both 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 can execute it using the GPM program. By interactive theorem proving we verify that the safety properties of agreement and validity (described below) are logical consequences of the LoE specification automatically generated for SC. That verification is described in this section and Sec. 5. Sec. 6 describes the process by which we automatically generate a GPM program and automatically verify that it implements the LoE specification, and is therefore an implementation of SC that provably satisfies both properties.

4.1. 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. 3. 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. The formal statements of these properties contain a universally quantified event ordering *eo* that the notation suppresses. Also, we assume additional constraints on *eo* and on SC's parameters that we discuss below in Sec. 4.2. Agreement says that notifications sent by SC never contradict ones another (the complete formal proof of the agreement property can be viewed at the following address: http://www.nuprl.org/LibrarySnapshots/Published/Version2/Event!ML/2!3!consensus!with!signatures/new_23_sig_agreement.html):

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

Validity says that any proposal decided on must be one that was proposed (the complete formal proof of the validity property can be viewed at the following address: http://www.nuprl.org/LibrarySnapshots/Published/Version2/Event!ML/2!3!consensus!with!signatures/new_23_sig_validity.html):

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

One subtlety: As mentioned above in Sec. 2.4, the reader can think of $\downarrow\exists$ as a classical existential. The squashing operator $\downarrow$, which enforces proof erasure, 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.

4.1.1. 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 $3 * \mathsf{F} + 1$ votes can be cast (from $3 * \mathsf{F} + 1$ replicas), and two different Quorum observers receive $2 * \mathsf{F} + 1$ 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 $2 * \mathsf{F} + 1$ votes

have been cast for c, then the majority of the votes in *any* collection of $2 * \mathsf{F} + 1$ 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 $k - 1$.

4.1.2. Proof of Validity

Validity is a corollary of the following lemma:

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

which says that any proposal contained in either a ``*decided*``, a ``*vote*``, or a ``*retry*`` message, was proposed at a causally prior event. We trivially prove this lemma by tracing back these messages to SC's inputs and outputs and eventually to proposals. For example, for a replica R_1 to receive a ``*retry*`` message, a replica R_2 has to have sent this ``*retry*`` message to R_1 and ``*retry*`` messages are only sent on receipt of a vote. To deduce this we use the assumption described next.

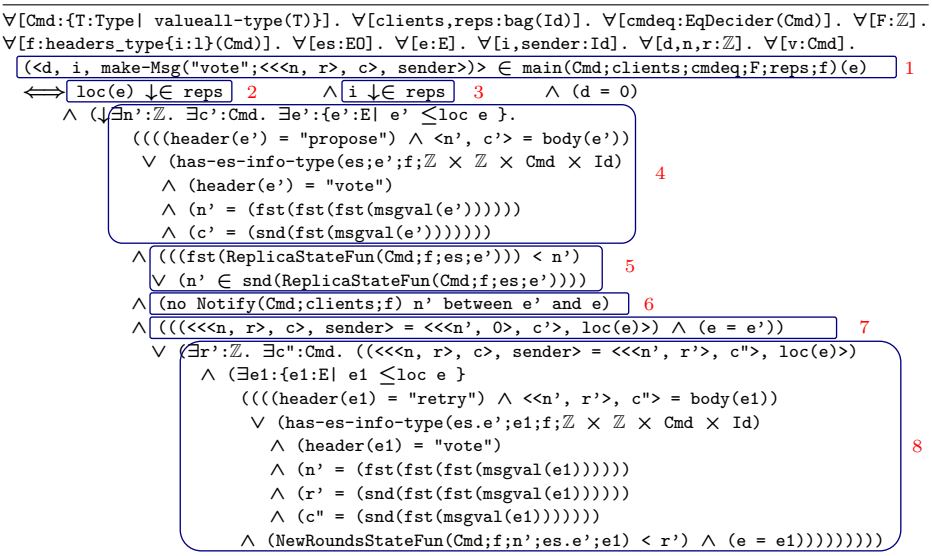
4.2. Assumptions

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, for any event ordering, if $v \in \mathsf{vote'base}(e)$, and e occurs at location *loc*, there must exist some $e' < e$ such that $(\mathsf{vote'send}\ \mathit{loc}\ v) \in \mathsf{SC}(e')$. Our tool does not enforce that the generated code respects such assumptions; therefore, they have to be enforced by other means. For example, the above assumption can be enforced, e.g., by physical means or by message encryption. We also assume that reps is a bag of size $3 * \mathsf{F} + 1$ without repetitions.

5. Automation

We have developed two main automation tools that help us prove properties of distributed systems. One is a rewriting tool that uses 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.

Figure 4 ILF instance for ``*vote*`` messages



5.1. 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 as described in Sec. 2.4.

Given a main observer X, we wrote a program that starts with a formula of the form $v \in X(e)$ and repeatedly rewrites it using equivalences such as the ones presented in Sec. 2.4 (see double equations 1 and 2), 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 each kind of messages that the system outputs. The ILF for ``*vote*`` messages in SC is shown in Fig. 4. For SC we generate characterizations of the sending of ``*vote*``, ``*retry*``, ``*decided*``, and ``*notify*`` messages. Intuitively, an ILF instance for a given kind of message K, provides a slice (expressed as a mathematical formula) of the LoE specification that corresponds to the output K only. Such slices allow us to analyze specifications without having to consider the entire code. For example, the Quorum part of Round is irrelevant to reasoning about votes.

Fig. 4 shows the ILF instance for ``*vote*`` messages as generated by Nuprl. (Note: `<_,_>` is Nuprl's pair constructor.) The details of this formula are not

critical for understanding our methodology. However, let us explain how it characterizes the sending of ``*vote*`` messages. This formula says that a vote of the form `<<<n,r>,c>,sender>` is sent by SC at event `e` to location `i` (see box 1) iff:

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

Using such formulas we can trace back a distributed system's outputs 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.

5.2. State Machine Properties

As mentioned in Sec. 3.3, one can define Moore machines in EventML using the State and Memory 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 State and Memory 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 proven a number of general lemmas by induction on causal order that provide simple sufficient conditions for verifying each of these kinds of properties for both State and Memory state machines. These lemmas typically require the user to prove that the transitions of the state machine satisfy some invariant—and sometimes to prove, in addition, a "base case" property of its initial state.

We have also 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 datatypes such as lists, integers, etc. Let us now describe and illustrate these four kinds of properties using the QuorumState and NewRoundsState state machines defined in Fig. 2 and Fig. 3 respectively. The latter keeps track of the current round number for each slot number.

5.2.1. Invariant

An invariant of a QuorumState state of the form (cmds,locs) is that locs has no repeats and has the 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 that we have already proved about Memory. 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. The same is true about our other state machine properties.

An invariant of NewRoundsState is that its state is a positive integer:

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

5.2.2. Ordering

Ordering properties express relations between two states. For example, if QuorumState observes (cmds1,locs1) at e_1 and (cmds2,locs2) at e_2 such that $e_1 \leq_{\mathtt{loc}} e_2$, then cmds1 and locs1 are final segments of cmds2 and locs2 respectively. We call that ordering property quorum_fseg, which we state as follows:

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

An ordering property of NewRoundsState is that rounds can only increase over time. We express this property in EventML as follows:

```
ordering rounds_inc on r1 then r2 in (NewRoundsState n) == r1 <= r2 ;;
```

5.2.3. Progress

The state round observed by NewRoundsState can only increase if RoundInfo observes a vote or a retry for a round number round' > round. We call such a property *progress*, because some progress has actually been made, i.e., the state has been updated.

For example, if NewRoundsState(n) observes round1 at e_1 and round2 at a strictly later event e_2, such that progress has been made between e_1 and e_2, then round1 < round2. Progress here is characterized by the RoundInfo observer: the property (n = n' ∧ round < round') has to be true at some event e between e_1 and e_2 such that RoundInfo observes ((n',round'),cmd) at e and NewRoundsState(n) observes round at e. This property says that if at event e we receive a round number round' (observed by RoundInfo) greater than our current round number round (as observed by NewRoundsState(n), then we update our round number to round', which means that our round number is going to increase between e_1 and e_2. 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 condition (round). n' = n /\ round < round'
  == round1 < round2 ;;
```

5.2.4. Memory

Memory properties say that state machines do not ignore inputs. For example, if NewRoundsState(n) observes round1 at e_1 and round2 at a strictly later event e_2, and RoundInfo observes ((n,round'),cmd) at e_1 then round' <= round2. We call that property, rounds_mem. It says that inputs characterized by RoundInfo are not ignored by NewRoundsState state machines.

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 ;;
```

5.3. 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). In terms of size: the LoE specification has a size of about 850 AST nodes, and the proof's size is about 8200 AST nodes. We only discuss safety properties here because we have not yet proved that 2/3 consensus is live/non-blocking.

6. 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. Both of these are automatically generated from an EventML specification. We carry out our correctness proofs on the LoE description of the main event observer. To gain trust in 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 and done automatically in Nuprl: 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 us consider the LoE parallel combinator defined in Sec. 2.4. Let X_1 and X_2 be event observers of type T, implemented by pr_1 and pr_2, respectively. The GPM parallel combinator $pr_1 \,||\, pr_2$ is defined as follows (for simplicity we use the same symbol as for the LoE parallel combinators):

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

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[5] p_i to m to obtain a new process p_i' and a bag of outputs 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.

As mentioned above, given a specification, EventML generates a GPM program, which is a mapping M (a λ-term) from locations to processes. Given a collection of locations L, one first applies M to each l in L to obtain all the involved local processes. As described in Sec. 1, a local process has type $\texttt{corec}(\lambda P.(A \to P \times \texttt{Bag}(B)) + \texttt{Unit})$ and runs on a single machine. For example, the GPM program generated by EventML from SC is a λ-expression that given a location l in reps returns a process that implements Replica at location l, and for all other locations not in reps returns the halted process. Therefore, once one has generated some code, one still needs to apply the generated GPM program to $3 * \mathsf{F} + 1$ replica locations, to obtain $3 * \mathsf{F} + 1$ local processes. Then, one can either execute these GPM processes, which are Nuprl terms, using Nuprl's

[5] 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 $(f\ m)$.

interpreter, i.e., directly using Nuprl's computation rules (Nuprl's computation system is an untyped λ-calculus, and as such one of its computation rules is the standard β-reduction rule); or one can use our Nuprl to Lisp translator and use a Lisp compiler to execute the code. Finally, EventML provides runtime environments to establish connections between nodes, and send and receive messages over the network. Therefore, the trusted code base of our system when using Nuprl's interpreter is: (1) Nuprl, (2) whichever Nuprl interpreter one decides to use (the one provided by default by EventML is implemented in SML), (3) the compiler used to compile the Nuprl interpreter, and (4) EventML's runtime environment to send and receive messages. When using our Nuprl to Lisp translator, the trusted code base is then: (1) Nuprl, (2) the Nuprl to lisp translator, (3), whichever Lisp compiler one decides to use, and (4) our Lisp runtime environment for EventML.

7. EventML Syntax and Semantics

This section defines EventML's syntax and static semantics but not its dynamic semantics because to run programs, one first has to compile EventML specifications to GPM processes and then, as mentioned above, use Nuprl's interpreter to execute these GPM processes.

Fig. 5 and Fig. 6 presents EventML's syntax. Note that the symbol ϵ is not part of the syntax but merely provides a visually convenient way of displaying the empty sequence of external types.

Next we introduce EventML's static semantics, which is similar to SML's as defined by Milner et al. [47]. The only forms for which we do not provide a semantics are (1) specification declarations of the form specification *vid* because they are only used to generate convenient names when loading an EventML specification into Nuprl; and (2) state machine properties because these are only used to automate the verification and Nuprl is there to catch type errors. In order to define the static semantics of the other forms, let us first define internal types, which are similar to the external types defined in Fig. 5. We do not distinguish between internal and external type constructor (postfix of the form *tc* and infix of the form *itc*). Type variables are shared by the internal and external syntax. In addition we introduce a new kind of type variable called an equality type variable. The type variables introduced in Fig. 5 are now called non-equality type variables. One can substitute a non-equality type variable (in set TyVar) by any type. However, one can only substitute an equality type variable by a type that has decidable equality, e.g., integers, Booleans, etc., but not functions—this is formally defined below. Internal types are defined as follows, where $\overrightarrow{\tau}$ is a sequence of internal types of the form $\langle \tau_1, \ldots, \tau_n \rangle$:

$$
\begin{array}{lll}
ea \in \mathsf{EqTyVar} & & \text{(set of equality type variables)} \\
\alpha \in \mathsf{ITyVar} & ::= a \mid ea & \\
\tau \in \mathsf{ITy} & ::= \alpha \mid \overrightarrow{\tau}\ tc \mid \tau_1\ itc\ \tau_2 &
\end{array}
$$

Type schemes and type environments are defined as follows, where $\overline{\alpha}$ is a set of type variables:

Figure 5 EventML syntax—expressions/patterns/types

n	$\in$ Nat		(natural numbers)
vid	$\in$ Vid		(infinite countable set of value identifiers)
tid	$\in$ Tid		(infinite countable set of type identifiers)
a	$\in$ TyVar		(infinite countable set of type variables)
tc	$\in$ TyCon	::=	tid | Int | Bool | Unit | Type | Loc | Interface
			| List | Bag | Deq | Obs
itc	$\in$ InfTyCon	::=	$*$ | $\rightarrow$ |+
opr	$\in$ Opr	::=	+ | - | = | . |++ | > |< | or | & | @ | || | >>= | /\ | \/ | =>
exp	$\in$ Exp	::=	vid | n | true | false | op vid
			| $(exp_1,\ldots,exp_n)$
			| $exp_1\ opr\ exp_2$
			| $exp{:}ty$
			| $::ty$
			| $exp_1\ exp_2$
			| $\backslash pat.exp$
			| if exp_1 then exp_2 else exp_3
			| let $bind$ in exp
			| exp where $bind$
			| $[exp_1,\ldots,exp_n]$
			| $\{exp_1,\ldots,exp_n\}$
			| exp o $(exp_1,\ldots,exp_n)$
			| Memory(exp_1,exp_2,exp_3)
			| State(exp_1,exp_2,exp_3)
			| Output(exp)
			| Once(exp)
			| exp_1 until exp_2
pat	$\in$ Pat	::=	vid | _ | $(pat_1,\ldots,pat_n)$ | $pat{:}ty$
$tyseq$	$\in$ TySeq	::=	ϵ | ty | $(ty_0,\ldots,ty_n)$
ty	$\in$ Ty	::=	a | $tyseq\ tc$ | $ty_1\ itc\ ty_2$ | (ty)

$$
\begin{array}{lll}
\sigma \in \mathsf{ITyScheme} & ::= & \forall \overline{\alpha}.\ \tau \\
e \in \mathsf{ITyEnvElt} & ::= & vid \mapsto \sigma \mid tid \mapsto \tau \\
\Gamma \in \mathsf{ITyEnv} & = & \{\ \Gamma \in \mathbb{P}(\mathsf{ITyEnvElt}) \\
 & & \mid\ (vid \mapsto \sigma_1, vid \mapsto \sigma_2 \in \Gamma \Rightarrow \sigma_1 = \sigma_2) \\
 & & \wedge\ (tid \mapsto \tau_1, tid \mapsto \tau_2 \in \Gamma \Rightarrow \tau_1 = \tau_2)\}
\end{array}
$$

Therefore, type environment are sets that can also be seen as partial functions. Let $\Gamma(vid)$ be σ if $vid \mapsto \sigma \in \Gamma$ and undefined otherwise. Similarly, let $\Gamma(tid)$ be τ if $tid \mapsto \tau \in \Gamma$ and undefined otherwise. Given an environment Γ of the form $\{vid_1 \mapsto \sigma_1, \ldots, vid_n \mapsto \sigma_n, tid_1 \mapsto \tau_1, \ldots, tid_m \mapsto \tau_m\}$, let $\mathsf{dom}(\Gamma) = \{vid_1, \ldots, vid_n, tid_1, \ldots, tid_m\}$. Type schemes are subject to α-conversion. An equality type variable can only be renamed into an equality type variable, and similarly for non-equality type variables. We sometimes write τ for the type

Figure 6 EventML syntax—binders/declarations/programs

$$
\begin{array}{rcl}
\mathit{bind} \in \mathsf{Bind} & ::= & \mathit{vid}\ \mathit{atpat}_1\ \cdots\ \mathit{atpat}_n = \mathit{exp} \\
\mathit{dec} \in \mathsf{Dec} & ::= & \mathsf{let}\ \mathit{bind};; \\
& | & \mathsf{observer}\ \mathit{bind};; \\
& | & \mathsf{parameter}\ \mathit{vid}_1, \mathit{vid}_2 : \mathit{ty}_1 * \mathit{ty}_2 \\
& | & \mathsf{parameter}\ \mathit{vid} : \mathit{ty} \\
& | & \mathsf{type}\ \mathit{vid} = \mathit{ty} \\
& | & \mathsf{internal}\ \mathit{vid} : \mathit{ty} \\
& | & \mathsf{input}\ \mathit{vid} : \mathit{ty} \\
& | & \mathsf{output}\ \mathit{vid} : \mathit{ty} \\
& | & \mathsf{import}\ \mathit{vid}_1 \ldots \mathit{vid}_n \\
& | & \mathsf{main}\ \mathit{exp} \\
& | & \mathsf{specification}\ \ \mathit{vid} \\
& | & \mathsf{invariant}\ \ \mathit{vid}\ \mathsf{on}\ \mathit{pat}\ \mathsf{in}\ \mathit{vid}_0\ \mathit{vid}_1\ \ldots\ \mathit{vid}_n == \mathit{exp};; \\
& | & \mathsf{ordering}\ \ \mathit{vid}\ \mathsf{on}\ \mathit{pat}_1\ \mathsf{then}\ \mathit{pat}_2\ \mathsf{in}\ \mathit{vid}_0\ \mathit{vid}_1\ \ldots\ \mathit{vid}_n == \mathit{exp};; \\
& | & \mathsf{progress}\ \ \mathit{vid}\ \mathsf{on}\ \mathit{pat}_1\ \mathsf{then}\ \mathit{pat}_2\ \mathsf{in}\ \mathit{vid}_0\ \mathit{vid}_1\ \ldots\ \mathit{vid}_n \\
& & \quad \mathsf{with}\ \ \mathit{pat}_3\ \mathsf{in}\ \mathit{vid}' \\
& & \quad\ \ \mathsf{and\ condition}\ \ (\mathit{pat}_4)\ .\ \mathit{exp}_1 \\
& & \quad == \mathit{exp}_2;; \\
& | & \mathsf{memory}\ \mathit{vid}\ \mathsf{on}\ \mathit{pat}_1\ \mathsf{then}\ \mathit{pat}_2\ \mathsf{in}\ \mathit{vid}_0\ \mathit{vid}_1\ \ldots\ \mathit{vid}_n \\
& & \quad \mathsf{with}\ \ \mathit{pat}_3\ \mathsf{in}\ \mathit{vid}' \\
& & \quad == \mathit{exp};; \\
\mathit{prog} \in \mathsf{Prog} & ::= & \mathit{dec} \mid \mathit{dec}\ \mathit{prog}
\end{array}
$$

scheme $\forall \varnothing.\ \tau$, and tc for the internal type $\langle\rangle\ tc$. Also, let:

$$\Gamma_1 + \Gamma_2 = \Gamma_2 \cup \begin{array}{l} \{\mathit{vid} \mapsto \sigma \mid \mathit{vid} \mapsto \sigma \in \Gamma_1 \wedge \mathit{vid} \notin \mathsf{dom}(\Gamma_2)\} \\ \{\mathit{tid} \mapsto \tau \mid \mathit{tid} \mapsto \tau \in \Gamma_1 \wedge \mathit{tid} \notin \mathsf{dom}(\Gamma_2)\} \end{array}$$

Let $\Gamma_1 \boxplus \Gamma_2$ be $\Gamma_1 + \Gamma_2$ if $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \varnothing$ and undefined otherwise.

We define the *admits equality* predicate on internal type constructors, internal types, and internal type sequences as follows (where Interface is the type of directed messages):

$$
\begin{array}{l}
\mathsf{admitsEq}(tc) \iff tc \notin \{\mathsf{Type}, \mathsf{Interface}, \mathsf{Obs}, \mathsf{Deq}\} \\
\mathsf{admitsEq}(\mathit{itc}) \iff \mathit{itc} \neq \rightarrow
\end{array}
$$

$$
\begin{array}{l}
\mathsf{admitsEq}(\tau) \iff \\
\qquad \tau = \mathit{ea} \\
\qquad \vee\ \tau = \overrightarrow{\tau}\ \ tc \wedge \mathsf{admitsEq}(tc) \wedge \mathsf{admitsEq}(\overrightarrow{\tau}) \\
\qquad \vee\ \tau = \tau_1\ \ \mathit{itc}\ \ \tau_2 \wedge \mathsf{admitsEq}(\mathit{itc}) \wedge \mathsf{admitsEq}(\langle \tau_1, \tau_2 \rangle) \\
\mathsf{admitsEq}(\langle \tau_1, \ldots, \tau_n \rangle) \iff \forall i \in \{1, \ldots, n\}.\ \mathsf{admitsEq}(\tau_i)
\end{array}
$$

We define substitutions as follows:

$$\mathit{sub} \in \mathsf{Sub} = \{f \in \mathsf{ITyVar} \to \mathsf{ITy} \mid \forall \mathit{ea}.\ \mathsf{admitsEq}(f(\mathit{ea}))\}$$

Substitutions are applied to internal types as follows:

$$
\begin{array}{ll}
\alpha[sub] & = sub(\alpha) \\
(\langle \tau_1, \ldots, \tau_n \rangle\ tc)[sub] & = \langle \tau_1[sub], \ldots, \tau_n[sub] \rangle\ tc \\
(\tau_1\ itc\ \tau_2)[sub] & = \tau_1[sub]\ itc\ \tau_n[sub]
\end{array}
$$

A type τ is an instance of a type scheme $\forall \overline{\alpha}.\ \tau'$, written $\tau \prec \forall \overline{\alpha}.\ \tau'$, iff there exists a substitution sub such that $\tau = \tau'[sub]$. We use this relation to define the static semantics of identifier expressions and of binary operators in Fig. 7.

We compute the set of free type variables of internal types and type environments as follows:

$$
\begin{array}{l}
\mathsf{fv}(\tau) = \{\alpha \mid \alpha \text{ occurs in } \tau\} \\
\mathsf{fv}(\Gamma) = \{\alpha \mid \Gamma(vid) = \forall \overline{\alpha}.\ \tau \wedge \alpha \in \mathsf{fv}(\tau) \setminus \overline{\alpha}\} \\
\qquad\qquad \cup\ \{\alpha \mid \Gamma(tid) = \tau \wedge \alpha \in \mathsf{fv}(\tau)\}
\end{array}
$$

Let the closure of a type environment be defined as follows:

$$
\mathsf{clos}_\Gamma(\Gamma') = \{vid \mapsto \forall(\mathsf{fv}(\tau) \setminus \mathsf{fv}(\Gamma)).\ \tau \mid \Gamma'(vid) = \tau\}
$$

We call closure of a type environment, the transformation of the monomorphic part of a type environment into a polymorphic one, promoting types to type schemes. In the above definition, we close the environment Γ' in the context of the type environment Γ. We use this function to define the static semantics of let expressions and let declarations in Fig. 7 and Fig. 10.

Let $\Gamma\mathsf{op}$ be the following environment for binary operators:

$$
\begin{array}{ll}
\{+ & \mapsto \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
, - & \mapsto \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\
, = & \mapsto \forall\{ea\}.\ ea \to ea \to \mathsf{Bool} \\
, . & \mapsto \forall\{a\}.\ a \to a\ \mathsf{List} \to a\ \mathsf{List} \\
, ++ & \mapsto \forall\{a\}.\ a\ \mathsf{List} \to a\ \mathsf{List} \to a\ \mathsf{List} \\
, < & \mapsto \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
, > & \mapsto \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \\
, \mathsf{or} & \mapsto \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
, \& & \mapsto \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
, @ & \mapsto \forall\{a\}.\ a\ \mathsf{Obs} \to \mathsf{Loc\ Bag} \to a\ \mathsf{Obs} \\
, || & \mapsto \forall\{a\}.\ a\ \mathsf{Obs} \to a\ \mathsf{Obs} \to a\ \mathsf{Obs} \\
, >>= & \mapsto \forall\{a, a'\}.\ a\ \mathsf{Obs} \to (a \to a'\ \mathsf{Obs}) \to a'\ \mathsf{Obs} \\
\} &
\end{array}
$$

The only special case in the definition of this initial environment is the `=` case, where we enforce that its arguments have to "admit equality". This prevents from using the equality binary operator on, e.g., functions.

Finally, Fig. 7 presents the static semantics of EventML's expressions. It defines the relation $(exp : \langle \Gamma, \tau \rangle)$. Fig. 8 presents the static semantics of EventML's patterns. It defines the relation $(pat :_{\mathrm{p}} \langle \Gamma, \tau \rangle)$. Fig. 9 presents the static semantics of EventML's external types. It defines the relations $(tyseq :_{\mathrm{s}} \langle \Gamma, \overrightarrow{\tau} \rangle)$ and $(ty :_{\mathrm{t}} \langle \Gamma, \tau \rangle)$. Fig. 10 presents the static semantics of EventML's bindings and declarations. It defines the relation $(x :_{\mathrm{d}} \langle \Gamma, \Gamma' \rangle)$ where x can either be a *bind*, or a *dec*, or a *prog*.

Figure 7 EventML's static semantics—expressions

$$\frac{\tau \prec \Gamma(vid)}{vid : \langle \Gamma, \tau \rangle} \qquad \frac{}{\mathsf{true} : \langle \Gamma, \mathsf{Bool} \rangle} \qquad \frac{}{\mathsf{false} : \langle \Gamma, \mathsf{Bool} \rangle} \qquad \frac{}{n : \langle \Gamma, \mathsf{Int} \rangle} \qquad \frac{}{() : \langle \Gamma, \mathsf{Unit} \rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ exp_i : \langle \Gamma, \tau_i \rangle}{(exp_1, \ldots, exp_n) : \langle \Gamma, \tau_1 \ * \ \cdots \ * \ \tau_n \rangle} \qquad \frac{exp_1 : \langle \Gamma, \tau_1 \to \tau_2 \rangle \quad exp_2 : \langle \Gamma, \tau_1 \rangle}{exp_1 \ exp_2 : \langle \Gamma, \tau_2 \rangle}$$

$$\frac{exp : \langle \Gamma, \tau \rangle \quad ty :_{\mathrm{t}} \langle \Gamma, \tau \rangle}{exp{:}ty : \langle \Gamma, \tau \rangle} \qquad \frac{ty :_{\mathrm{t}} \langle \Gamma, \tau \rangle}{::\, ty : \langle \Gamma, \mathsf{Type} \rangle} \qquad \frac{pat :_{\mathrm{p}} \langle \Gamma', \tau \rangle \quad exp : \langle \Gamma + \Gamma', \tau' \rangle}{\backslash pat.exp : \langle \Gamma, \tau \to \tau' \rangle}$$

$$\frac{exp_1 : \langle \Gamma, \tau_1 \rangle \quad exp_2 : \langle \Gamma, \tau_2 \rangle \quad (\tau_1 \to \tau_2 \to \tau_3) \prec \Gamma(opr)}{exp_1 \ opr \ exp_2 : \langle \Gamma, \tau_3 \rangle} \qquad \frac{(\tau_1 \to \tau_2 \to \tau_3) \prec \Gamma(vid)}{\mathsf{op}\ vid : \langle \Gamma, \tau_1 \to \tau_2 \to \tau_3 \rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ exp_i : \langle \Gamma, \tau \rangle}{[exp_1, \ldots, exp_n] : \langle \Gamma, \tau \ \mathsf{List} \rangle} \qquad \frac{exp_1 : \langle \Gamma, \mathsf{Bool} \rangle \quad exp_2 : \langle \Gamma, \tau \rangle \quad exp_3 : \langle \Gamma, \tau \rangle}{\mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3 : \langle \Gamma, \tau \rangle}$$

$$\frac{bind :_{\mathrm{d}} \langle \Gamma, \Gamma' \rangle \quad exp : \langle \Gamma + \mathsf{clos}_\Gamma(\Gamma'), \tau \rangle}{\mathsf{let}\ \ bind\ \mathsf{in}\ exp : \langle \Gamma, \tau \rangle} \qquad \frac{bind :_{\mathrm{d}} \langle \Gamma, \Gamma' \rangle \quad exp : \langle \Gamma + \mathsf{clos}_\Gamma(\Gamma'), \tau \rangle}{exp\ \mathsf{where}\ bind : \langle \Gamma, \tau \rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ exp_i : \langle \Gamma, \tau \rangle}{\{exp_1, \ldots, exp_n\} : \langle \Gamma, \tau \ \mathsf{Bag} \rangle} \qquad \frac{\begin{array}{l} exp : \langle \Gamma, \mathsf{Loc} \to \tau_1 \to \cdots \to \tau_n \to \tau_{n+1} \rangle \\ \forall i \in \{1, \ldots, n\}.\ exp_i : \langle \Gamma, \tau_i\ \mathsf{Obs} \rangle \end{array}}{exp\ \mathsf{o}\ (exp_1, \ldots, exp_n) : \langle \Gamma, \tau_{n+1}\ \mathsf{Obs} \rangle}$$

$$\frac{\begin{array}{l} exp_1 : \langle \Gamma, \mathsf{Loc} \to \tau_2 \rangle \\ exp_2 : \langle \Gamma, \mathsf{Loc} \to \tau_1 \to \tau_2 \to \tau_2 \rangle \\ exp_3 : \langle \Gamma, \tau_1\ \mathsf{Obs} \rangle \end{array}}{\mathsf{Memory}(exp_1, exp_2, exp_3) : \langle \Gamma, \tau_2\ \mathsf{Obs} \rangle} \qquad \frac{\begin{array}{l} exp_1 : \langle \Gamma, \mathsf{Loc} \to \tau_2 \rangle \\ exp_2 : \langle \Gamma, \mathsf{Loc} \to \tau_1 \to \tau_2 \to \tau_2 \rangle \\ exp_3 : \langle \Gamma, \tau_1\ \mathsf{Obs} \rangle \end{array}}{\mathsf{State}(exp_1, exp_2, exp_3) : \langle \Gamma, \tau_2\ \mathsf{Obs} \rangle}$$

$$\frac{exp : \langle \Gamma, \mathsf{Loc} \to \tau\ \mathsf{Bag} \rangle}{\mathsf{Output}(exp) : \langle \Gamma, \tau\ \mathsf{Obs} \rangle} \qquad \frac{exp : \langle \Gamma, \tau\ \mathsf{Obs} \rangle}{\mathsf{Once}(exp) : \langle \Gamma, \tau\ \mathsf{Obs} \rangle}$$

$$\frac{exp_1 : \langle \Gamma, \tau\ \mathsf{Obs} \rangle \quad exp_2 : \langle \Gamma, \tau'\ \mathsf{Obs} \rangle}{exp_1\ \mathsf{until}\ exp_2 : \langle \Gamma, \tau\ \mathsf{Obs} \rangle}$$

Figure 8 EventML's static semantics—patterns

$$\frac{}{vid :_{\mathrm{p}} \langle \{vid \mapsto \tau\}, \tau \rangle} \qquad \frac{}{\_ :_{\mathrm{p}} \langle \{\}, \tau \rangle} \qquad \frac{pat :_{\mathrm{p}} \langle \Gamma, \tau \rangle \quad ty :_{\mathrm{t}} \langle \Gamma, \tau \rangle}{pat{:}ty :_{\mathrm{p}} \langle \Gamma, \tau \rangle} \qquad \frac{}{() :_{\mathrm{p}} \langle \varnothing, \mathsf{Unit} \rangle}$$

$$\frac{\forall i \in \{1, \ldots, n\}.\ pat_i :_{\mathrm{p}} \langle \Gamma_i, \tau_i \rangle}{(pat_1, \ldots, pat_n) :_{\mathrm{p}} \langle \Gamma_1 \boxplus \cdots \boxplus \Gamma_n, \tau_1 \ * \ \cdots \ * \ \tau_n \rangle}$$

A piece of code *prog* is a valid EventML program iff there exists a type environment Γ such that $prog :_{\mathrm{d}} \langle \Gamma\mathsf{op}, \Gamma \rangle$ can be derived from the rules presented in

Figure 9 EventML's static semantics—types

$$\frac{}{\epsilon :_{\mathrm{s}} \langle \Gamma, \langle\rangle\rangle} \qquad \frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle}{ty :_{\mathrm{s}} \langle \Gamma, \langle\tau\rangle\rangle} \qquad \frac{\forall i \in \{0, \ldots, n\}.\ ty_i :_{\mathrm{t}} \langle \Gamma, \tau_i\rangle}{(ty_0, \ldots, ty_n) :_{\mathrm{s}} \langle \Gamma, \langle\tau_0, \ldots, \tau_n\rangle\rangle}$$

$$\frac{}{a :_{\mathrm{t}} \langle \Gamma, a\rangle} \qquad \frac{tyseq :_{\mathrm{s}} \langle \Gamma, \overrightarrow{\tau}\rangle}{tyseq\ tc :_{\mathrm{t}} \langle \Gamma, \overrightarrow{\tau}\ tc\rangle} \qquad \frac{ty_1 :_{\mathrm{t}} \langle \Gamma, \tau_1\rangle \quad ty_2 :_{\mathrm{t}} \langle \Gamma, \tau_2\rangle}{ty_1\ itc\ ty_2 :_{\mathrm{t}} \langle \Gamma, \tau_1\ itc\ \tau_2\rangle} \qquad \frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle}{(ty) :_{\mathrm{t}} \langle \Gamma, \tau\rangle}$$

Fig. 7 to Fig. 10. The environment Γ is *prog*'s static semantics. It corresponds to the well-formedness lemmas generated when loading an EventML specification into Nuprl.

8. Related Work

Much work has been done on specifying and reasoning about distributed systems [41, 27, 16, 19, 20, 16, 18, 30, 67, 29, 68] (to only cite a few).

Event-B. Event-B [1] is a set-theory-based language for modeling reactive systems and deriving low-level (concrete, i.e. referring to implementation details) specifications from high-level (abstract) specifications—this is usually known as *refinement*. Event-B is supported by the Rodin platform [2], which provides support for refinement as well as both automated and interactive theorem proving. It has been used to verify a wide variety of systems and supports code generation [46, 24]. For example, Bryans [16] used Event-B to verify the key properties of a synchronous crash-tolerant consensus algorithm called Floodset [44]. His refinement method allows him to prove invariants "at the earliest possible stage, before the introduction of distracting detail." Bryans proved that Floodset satisfies agreement and validity, and in addition he showed using ProB—a model-checker for Event-B—that the algorithm terminates. Event-B has also been used to model and verify safety and liveness properties of self-$\star$ systems [6] (including self-healing, self-stabilizing, self-organizing, etc.). The authors illustrate their methodology using a P2P-based self-healing protocol. Using the refinement method supported by Event-B, they produced models that are close to running code, and left "generating applications from the resulting model" for future work. As opposed to this work, we focus here on a crash-fault model of distributed systems and on producing running code. Moreover, as opposed to the standard refinement method, in our methodology we rely on the fact that our DSL is based on a small core of combinators for which we know that they can be interpreted (in a single formal method tool) both by logical forms to perform the verification and by running code that provably implement the logical forms.

IOA. IOA [26, 13, 25, 63, 27] is a programming/specification language for describing asynchronous distributed systems as I/O automata [45] (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,

Figure 10 EventML's static semantics—bindings and declarations

$$\frac{\forall i \in \{1,\ldots,n\}.\ atpat_i :_{\mathrm{p}} \langle \Gamma_i, \tau_i\rangle \quad exp : \langle \Gamma + (\Gamma_1 \boxplus \cdots \boxplus \Gamma_n), \tau\rangle}{vid\ atpat_1\ \cdots\ atpat_n = exp :_{\mathrm{d}} \langle \Gamma, \{vid \mapsto \tau_1 \to \cdots \to \tau_n \to \tau\}\rangle}$$

$$\frac{bind :_{\mathrm{d}} \langle \Gamma, \Gamma'\rangle}{\mathsf{let}\ \ bind;; :_{\mathrm{d}} \langle \Gamma, \mathsf{clos}_\Gamma(\Gamma')\rangle}$$

$$\frac{bind :_{\mathrm{d}} \langle \Gamma, \{vid \mapsto \tau_1 \to \cdots \to \tau_n \to \tau\ \mathsf{Obs}\}\rangle}{\mathsf{observer}\ \ bind;; :_{\mathrm{d}} \langle \Gamma, \mathsf{clos}_\Gamma(\{vid \mapsto \tau_1 \to \cdots \to \tau_n \to \tau\ \mathsf{Obs}\})\rangle}$$

$$\frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle \quad \mathsf{fv}(\tau) = \varnothing}{\mathsf{parameter}\ vid : ty :_{\mathrm{d}} \langle \Gamma, \{vid \mapsto \tau\}\rangle}$$

$$\frac{ty_1 :_{\mathrm{t}} \langle \Gamma, \tau_1\rangle \quad ty_2 :_{\mathrm{t}} \langle \Gamma, \tau_2\rangle \quad \mathsf{fv}(\tau_1) \cup \mathsf{fv}(\tau_2) = \varnothing}{\mathsf{parameter}\ vid_1,\ vid_2 : ty_1 * ty_2 :_{\mathrm{d}} \langle \Gamma, \{vid_1 \mapsto \tau_1, vid_2 \mapsto \tau_2\}\rangle}$$

$$\frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle}{\mathsf{type}\ vid = ty :_{\mathrm{d}} \langle \Gamma, \{vid \mapsto \tau\}\rangle}$$

$$\frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle \quad \Gamma' = \left\{ \begin{array}{l} vid\text{'send} \mapsto \mathsf{Loc} \to \tau \to\ \mathsf{Interface}\ , \\ vid\text{'bcast} \mapsto \mathsf{Loc} \to \tau \to\ \mathsf{Interface}\ \mathsf{Bag}, \\ vid\text{'base} \mapsto \tau\ \mathsf{Obs} \end{array} \right\}}{\mathsf{internal}\ \ vid : ty :_{\mathrm{d}} \langle \Gamma, \Gamma'\rangle}$$

$$\frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle \quad \Gamma' = \{vid\text{'base} \mapsto \tau\ \mathsf{Obs}\}}{\mathsf{input}\ \ vid : ty :_{\mathrm{d}} \langle \Gamma, \Gamma'\rangle}$$

$$\frac{ty :_{\mathrm{t}} \langle \Gamma, \tau\rangle \quad \Gamma' = \left\{ \begin{array}{l} vid\text{'send} \mapsto \mathsf{Loc} \to \tau \to\ \mathsf{Interface}\ , \\ vid\text{'bcast} \mapsto \mathsf{Loc} \to \tau \to\ \mathsf{Interface}\ \mathsf{Bag} \end{array} \right\}}{\mathsf{output}\ \ vid : ty :_{\mathrm{d}} \langle \Gamma, \Gamma'\rangle}$$

$$\frac{\forall i \in \{1,\ldots,n\}.\ \Gamma(vid_i) = \sigma_i}{\mathsf{import}\ \ vid_1 \ldots vid_n :_{\mathrm{d}} \langle \Gamma, \{vid_1 \mapsto \sigma_1\} + \cdots + \{vid_n \mapsto \sigma_n\}\rangle}$$

$$\frac{exp : \langle \Gamma,\ \mathsf{Interface}\ \mathsf{Obs}\rangle}{\mathsf{main}\ exp :_{\mathrm{d}} \langle \Gamma, \varnothing\rangle} \qquad \frac{dec :_{\mathrm{d}} \langle \Gamma, \Gamma'\rangle \quad \ulcorner prog :_{\mathrm{d}} \langle \Gamma + \Gamma', \Gamma''\rangle \urcorner}{dec\ \ulcorner prog \urcorner :_{\mathrm{d}} \langle \Gamma, \Gamma' \ulcorner + \Gamma'' \urcorner\rangle}$$

and there is support for synthesis of Java code [63]. 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$^+$ [36, 20] is a language for specifying and reasoning about systems, that combines (1) TLA [38], which is a temporal logic that "provides a mathematical foundation for describing systems" [36], and (2) set theory, which is used there to specify data structures. TLAPS "is a platform for the development and mechanical verification of TLA$^+$ proofs" [20]. 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. TLA$^+$ has been used in a large number of projects such as [43, 31, 15, 50, 51, 8, 7] to cite only a few. 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.

Orc. Orc [33, 34, 32] is a programming language for structured concurrent programming. It is based on a small set of combinators to "orchestrate the concurrent invocations of sites" [33] that perform basic services (such as timers). Expressions in Orc are similar to our event observers. Among other things, Orc has similar parallel and delegation combinators and allows recursive definitions. However, to the best of our knowledge, Orc does not support program verification.

seL4. Our approach is similar to the one taken by Klein et al. to verify the seL4 microkernel [35]. 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.

FVN. Another similar approach to ours is the one taken by the FVN framework [66]; 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 [65]. This would correspond to the LoE part of our framework. Using P2 [42], NDlog programs can also be compiled to dataflow programs. This would correspond to GPM's level.

Verdi. More recently, Wilcox et al. developed Verdi [67], 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, Raft [52]—an alternative of Paxos—

transforms a distributed system into a crash-tolerant distributed system. 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. In [68], the authors report that they have completely verified Raft's safety. They also present their methodology to deal with proof maintenance, such as using interface lemmas in order to hide definitions [68, Sec.4].

IronFleet. IronFleet [29] is a framework to build and reason about distributed systems using Dafny [40], which is a programming and verification framework that relies on the Z3 theorem prover [49]. Because systems are both implemented in and verified using Dafny, IronFleet also does not have a gap between the code that is verified and the code that runs. The authors use a combination of TLA-style state-machine refinements [36] to reason about the distributed aspects of protocols, and Floyd-Hoare-style imperative verification techniques to reason about the local behavior of state machines. The authors have implemented and verified, among other things, IronRSL, which is a Paxos-based state machine replication library, of which they have proved both its safety and liveness.

9. Conclusion, Current and Future Work

Our methodology scales to more complicated and subtle distributed protocols: we have specified the Multi-Paxos protocol [37, 57] 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 [61, 56, 60]. To get efficient code, we have built in Nuprl a formal tool tuned to automatically optimize GPM programs and prove that the optimized and non-optimized programs are bisimilar [54]. 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 2/3 consensus (similar proofs have been carried out by Charron-Bost, Merz and Debrat [19, 18]). However, it turns out that these proofs are tedious. Next, we want to build automation tools to assist us in proving such properties.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: an open toolset for modelling and reasoning in Event-B". In: *International Journal on Software Tools for Technology Transfer (STTT)* 12.6 (2010), pp. 447–466.

[3] 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". In: *ECEASST* 35 (2010).

[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". In: *J. Applied Logic* 4.4 (2006). `http://www.nuprl.org/`, pp. 428–469.

[5] Abhishek Anand and Ross A. Knepper. "ROSCoq: Robots Powered by Constructive Reals". In: *Interactive Theorem Proving - 6th Int'l Conf., ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 34–50.

[6] Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. "Analysis of Self-$\star$ and P2P Systems Using Refinement". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th Int'l Conf., ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 117–123.

[7] Selma Azaiez, Damien Doligez, Matthieu Lemerre, Tomer Libal, and Stephan Merz. "Proving Determinacy of the PharOS Real-Time Operating System". In: *5th Int'l Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*. Vol. 9675. Lecture Notes in Computer Science. Springer, 2016, pp. 70–85.

[8] Noran Azmy, Stephan Merz, and Christoph Weidenbach. "A Rigorous Correctness Proof for Pastry". In: *5th Int'l Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*. Vol. 9675. Lecture Notes in Computer Science. Springer, 2016, pp. 86–101.

[9] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. `http://www.labri.fr/perso/casteran/CoqArt`. SpringerVerlag, 2004.

[10] Mark Bickford. "Component Specification Using Event Classes". In: *Component-Based Software Engineering, 12th Int'l Symp.* Vol. 5582. Lecture Notes in Computer Science. Springer, 2009, pp. 140–155.

[11] Mark Bickford, Robert L. Constable, and Vincent Rahli. "Logic of Events, a framework to reason about distributed systems". Presented at the Languages for Distributed Algorithms Workshop, Philadelphia, PA. 2012.

[12] Mark Bickford, Robert Constable, and David Guaspari. *Generating Event Logics with Higher-Order Processes as Realizers*. Tech. rep. Cornell University, 2010.

[13] Andrej Bogdanov. "Formal verification of simulations between I/O automata". MA thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[14] Péter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. "Efficient model checking of fault-tolerant distributed protocols". In: *Proceedings of the 2011 IEEE/IFIP Int'l Conf. on Dependable Systems and Networks, DSN 2011, Hong Kong, China, June 27-30 2011*. IEEE Compute Society, 2011, pp. 73–84.

[15] William J. Bolosky, John R. Douceur, and Jon Howell. "The Farsite project: a retrospective". In: *Operating Systems Review* 41.2 (2007), pp. 17–26.

[16] Jeremy W. Bryans. "Developing a Consensus Algorithm Using Stepwise Refinement". In: *Formal Methods and Software Engineering - 13th Int'l Conf. on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*. Vol. 6991. Lecture Notes in Computer Science. Springer, 2011, pp. 553–568.

[17] B. Charron-Bost and A. Schiper. "The Heard-Of model: computing in distributed systems with benign failures". In: *Distributed Computing* 22.1 (2009), pp. 49–71.

[18] Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. "Formal Verification of Consensus Algorithms Tolerating Malicious Faults". In: *Stabilization, Safety, and Security of Distributed Systems - 13th Int'l Symp., SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 120–134.

[19] Bernadette Charron-Bost and Stephan Merz. "Formal Verification of a Consensus Algorithm in the Heard-Of Model". In: *Int. J. Software and Informatics* 3.2-3 (2009), pp. 273–303.

[20] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. "Verifying Safety Properties with the TLA+ Proof System". In: *Automated Reasoning, 5th Int'l Joint Conf., IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Vol. 6173. Lecture Notes in Computer Science. Springer, 2010, pp. 142–148.

[21] 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*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

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

[23] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (1985), pp. 374–382.

[24] Andreas Fürst, Thai Son Hoang, David A. Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. "Code Generation for Event-B". In: *Integrated Formal Methods - 11th Int'l Conf., IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*. Vol. 8739. Lecture Notes in Computer Science. Springer, 2014, pp. 323–338.

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

[26] Stephen J. Garland and Nancy Lynch. "Using I/O automata for developing distributed systems". In: *Foundations of component-based systems*. New York, NY, USA: Cambridge University Press, 2000, pp. 285–312.

[27] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. "Automated implementation of complex distributed algorithms specified in the IOA language". In: *Int. J. Softw. Tools Technol. Transf.* 11 (2 Feb. 2009), pp. 153–171.

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

[29] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct". In: *Proceedings of the 25th Symp. on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 2015, pp. 1–17.

[30] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. "Parameterized model checking of fault-tolerant distributed algorithms by abstraction". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 201–209.

[31] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark R. Tuttle, and Yuan Yu. "Checking Cache-Coherence Protocols with TLA$^+$". In: *Formal Methods in System Design* 22.2 (2003), pp. 125–131.

[32] David Kitchin. "Orchestration and Atomicity". PhD thesis. The University of Texas at Austin, Aug. 2013.

[33] 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., Bonn, Germany, August 27-30, 2006, Proceedings*. Vol. 4137. Lecture Notes in Computer Science. Springer, 2006, pp. 477–491.

[34] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. "The Orc Programming Language". In: *Formal Techniques for Distributed Systems*. Vol. 5522. Lecture Notes in Computer Science. Springer, 2009, pp. 1–25.

[35] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the 22nd ACM Symp. on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009, pp. 207–220.

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

[37] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169.

[38] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923.

[39] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[40] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th Int'l Conf., LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. Vol. 6355. Lecture Notes in Computer Science. Springer, 2010, pp. 348–370.

[41] Patrick Lincoln and John M. Rushby. "A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model". In: *Digest of Papers: FTCS-23, The Twenty-Third Annual Int'l Symp. on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993*. IEEE Computer Society, 1993, pp. 402–411.

[42] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. "Implementing declarative overlays". In: *Proceedings of the 20th ACM Symp. on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. ACM, 2005, pp. 75–90.

[43] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. "Towards Verification of the Pastry Protocol Using TLA $^+$ ". In: *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 Int'l Conf., FMOODS 2011, and 31st IFIP WG 6.1 Int'l Conf., FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*. Vol. 6722. Lecture Notes in Computer Science. Springer, 2011, pp. 244–258.

[44] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[45] Nancy A. Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *Proceedings of the Sixth Annual ACM Symp. on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*. ACM, 1987, pp. 137–151.

[46] Dominique Méry and Neeraj Kumar Singh. "Automatic code generation from event-B models". In: *Proceedings of the 2011 Symp. on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011*. ACM, 2011, pp. 179–188.

[47] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. Cambridge, MA, USA: MIT Press, 1997.

[48] Eugenio Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings, Fourth Annual Symp. on Logic in Computer Science, 5-8 June, 1989, Asilomar Conference Center, Pacific Grove, California, USA*. IEEE Computer Society, 1989, pp. 14–23.

[49] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th Int'l Conf., TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[50] Chris Newcombe. "Why Amazon Chose TLA +". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th Int'l Conf., ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. Vol. 8477. Lecture Notes in Computer Science. Springer, 2014, pp. 25–39.

[51] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon web services uses formal methods". In: *Commun. ACM* 58.4 (2015), pp. 66–73.

[52] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 2014, pp. 305–319.

[53] Vincent Rahli. "Interfacing with Proof Assistants for Domain Specific Programming Using EventML". Presented at the 10th International Workshop on User Interfaces for Theorem Provers. 2012.

[54] Vincent Rahli, Mark Bickford, and Abhishek Anand. "Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types". In: *Interactive Theorem Proving - 4th Int'l Conf., ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Vol. 7998. Lecture Notes in Computer Science. Springer, 2013, pp. 261–278.

[55] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. "Formal Specification, Verification, and Implementation of Fault-Tolerant Systems using EventML". In: *ECEASST* 72 (2015). Presented at AVoCS 2015.

[56] Vincent Rahli, Nicolas Schiper, Robbert van Renesse, Mark Bickford, and Robert L. Constable. "A diversified and correct-by-construction broadcast service". In: *20th IEEE International Conference on Network Protocols, ICNP 2012, Austin, TX, USA, October 30 - Nov. 2, 2012*. IEEE Computer Society, 2012, pp. 1–6.

[57] Robbert van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* 47.3 (2015), 5:1–5:36.

[58] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[59] Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. "Efficient Verification of Distributed Protocols Using Stateful Model Checking". In: *IEEE 32nd Symp. on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 2013, pp. 133–142.

[60] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. "Developing Correctly Replicated Databases Using Formal Tools". In: *44th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*. IEEE, 2014, pp. 395–406.

[61] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. "ShadowDB: A Replicated Database on a Synthesized Consensus Core". In: *Proceedings of the Eighth Workshop on Hot Topics in System Dependability, HotDep 2012, Hollywood, CA, USA, October 7, 2012*. USENIX Association, 2012.

[62] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319.

[63] 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.

[64] Tatsuhiro Tsuchiya and André Schiper. "Using Bounded Model Checking to Verify Consensus Algorithms". In: *Distributed Computing, 22nd Int'l Symp., DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*. Vol. 5218. Lecture Notes in Computer Science. Springer, 2008, pp. 466–480.

[65] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. "Declarative Network Verification". In: *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*. Vol. 5418. Lecture Notes in Computer Science. Springer, 2009, pp. 61–75.

[66] Anduo Wang, Limin Jia, Changbin Liu, Boon Thau Loo, Oleg Sokolsky, and Prithwish Basu. "Formally Verifiable Networking". In: *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS '09, New York City, NY, USA, October 22-23, 2009*. ACM SIGCOMM, 2009.

[67] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015, pp. 357–368.

[68] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. "Planning for change in a formal verification of the raft consensus protocol". In: *Proceedings of the 5th ACM SIGPLAN Conf. on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. ACM, 2016, pp. 154–165.