# Introduction to EventML

Mark Bickford, Robert L. Constable, Richard Eaton,
David Guaspari, and Vincent Rahli

November 24, 2011

# Contents

# 1 Introduction

## 1.1 Specification and Programming

EventML is a functional programming language in the ML family, closely related to Classic ML [GMW79, CHP84, KR11]. It is also a language for coding distributed protocols (such as Paxos [Ren11]) using high level *combinators* from the Logic of Events (or Event Logic) [Bic09, BC08], hence the name "EventML". The Event Logic combinators are high level specifications of distributed computations whose properties can be naturally expressed in Event Logic. The combinators can be formally translated (compiled) into the process model underlying Event Logic and thus converted to distributed programs. The interactions of these high level distributed programs manifest the behavior described by the logic. EventML can thus both specify and execute the processes that create the behaviors, called *event structures*, arising from the interactions of the processes.

Since EventML can directly specify computing tasks using the event combinators it can carry out part of the task normally assigned to a theorem prover, formal specification. EventML can also interact with a theorem prover, presently Nuprl [CAB+86, Kre02, ABC+06] (a theorem prover based on a constructive type theory called Computational Type Theory (CTT) [CAB+86] and on Classic ML), which can express logical properties and constraints on the evolving computations as formulas of Event Logic and prove them. From these proofs, a prover can create *correct-by-construction* process terms which EventML can execute. Thus EventML and Nuprl can work together synergistically in creating a correct by construction concurrent system. EventML could play the same role with respect to any theorem prover that implements the Logic of Events. Thus EventML provides a new paradigm for creating correct distributed systems, one in which a systems programmer can design and code a system using event combinators in such a way that a theorem prover can easily express and prove logical properties of the resulting computations. To EventML, the event combinators have a dual character. They have the *logical character* of specifications and the *computational character* of producing event structures with formally guaranteed behaviors.

## 1.2 Interaction with theorem provers

EventML was created to work in *cooperation* with an interactive theorem prover and to be a key component of a *Logical Programming Environment* (LPE) [ABC+06].

In one direction, EventML can import logical specifications from the prover as well as event class specifications and the process code that realizes them. In the present mode of operation, EventML *docks* with the Nuprl prover to obtain this information.

In the other direction, EventML can be used by programmers to specify protocols using event logic combinators. Following the line of work in which Nuprl was used to reason about the Ensemble system [Hay98, BCH+00, KHH98, LKvR+99] (coded in OCaml [Ler00]), EventML, by docking to Nuprl, provides a way to reason about (and synthesize) many distributed protocols. Thanks to its constructive logic, its expressiveness, and its large library, Nuprl is well suited to reason about distributed systems [BKR01]. But in principle EventML can connect to any prover that implements Event Logic and our General Process Model [BCG10]. Given an EventML specification, the Nuprl prover can: (1) synthesize process code, and (2) generate the *inductive logical form* of the specification which is used to structure logical description of the protocols and the system.

# 2 Event Logic

## 2.1 Events, event orderings, and event classes

The Logic of Events [Bic09, BC08] is a logic inspired by the work of Winskel on event structures [Win88], developed to deal with: (1) events; (2) their spatial locations; and (3) their temporal locations obtained via a well-founded ordering of these events (i.e., a temporal ordering). An event is triggered by receipt of

a message; the data of the message body is called *primitive information* of the event. The Logic of Events provides ways to describe events by, among other things giving access to their associated information.

An *event ordering* is a structure consisting of: (1) a set of events, (2) a location function `loc` that associates a *location* with each event, (3) an information function `info` that associates primitive information with each event, and (4) a well-founded *causal ordering* relation on events $<$ [Lam78]. An event ordering corresponds to a single run of a distributed system.

A basic concept in the Logic of Events is an *event class* [Bic09], which effectively partitions the events of an event ordering into those it "recognizes" and those it does not, and associates values to the events it recognizes. Different classes may recognize the same event and assign it different values. For example, one class may recognize the arrival of a message and associate it with its primitive information, the message data. Another class may recognize that, in the context of some protocol, the arrival of that message signifies successful completion of the protocol and assign to it a value meaning "send the 'success' message." We specify a concurrent system in EventML by defining event classes that appropriately classify system events.

Event classes have two facets: a programming one and a logical one. On the logical side, event classes specify information flow on a network of reactive agents by observing the information computed by the agents when events occur, i.e., on receipt of messages. On the programming side, event classes can be seen as processes that aggregate information in an internal state from input messages and past observations, and compute appropriate values for them.

Formally, an event class $X$ is a function whose inputs are event ordering and an event, and whose output is a bag of values (observations). If the observations are of type $T$, then the class $X$ is called an event class of type $T$. The associated type constructor is $\texttt{Class}(T) = \texttt{EO} \to \texttt{E} \to \texttt{Bag}(T)$, where `EO` is the type of event orderings and `E` the type of events. We reason about observations in terms of the *event class relation*: we say that $v$ is observed by the class $X$ at event $e$ (in an event ordering $eo$), and write $v \in X(e)$, if $v$ is a member of the bag $(X\ eo\ e)$. In our discussions, the relevant $eo$ will be clear from context, so our notation omits it. If the bag is nonempty we say that event $e$ is *in* the class $X$, and that $e$ is an $X$-event.

Event classes are ultimately defined from one kind of primitive event class (a *base class*) using six primitive *class combinators*—though users can define new combinators, and we supply a useful library of them. These primitives, and a variety of useful defined combinators are introduced in the examples of section 3.

### 2.2 Inductive logical forms

The inductive logical form of a specification is a first order formula that characterizes completely the observations (the responses) made by the main class of the specification in terms of the event class relation. The formula is inductive because it typically characterizes the responses at event $e$ in terms of observations made by a sub-component at a prior event $e' < e$. Such inductive logical forms are automatically generated in Nuprl from event class definitions, and simplified using various rewritings. From an inductive logical form we can prove invariants of the specification by induction on causal order.

## 3 Examples

We guide the reader through the features of this new programming/specification language with a series of examples.

### 3.1 Ping-pong

We consider the following setup: a client wants to run a protocol involving a certain collection of nodes, but first wants to know which of them are still alive. To learn that, the client initiates the ping-pong

**Figure 1** Ping-pong protocol

```
specification ping_pong

(* ------- Imported Nuprl definitions -------  *)
import bag-map;;

(* ------- Protocol parameters ------- *)
parameter p : Loc;;
parameter locs : Loc Bag;;

(* ------- Messages ------- *)
MSGS
  input    (``start`` : Loc, base Start)
  internal (``ping``  : Loc, base Ping,  send ping)
  internal (``pong``  : Loc, base Pong,  send pong)
  output   (``out``   : Loc, send out)
;;

(* ------- Classes ------- *)
class ReplyToPong (client,loc) =
  let F _ l = if l = loc then {out client loc} else {} in
    Once(F o Pong) ;;
class Ping (_,loc) = Output(\l.{ping loc l}) ;;
class Handler (c,l) = (SendPing (c,l) || ReplyToPong (c,l)) ;;

class P = ((\_.\client.bag-map (\l.(client,l)) locs) o Start) >>= Handler;;

class ReplyToPing = (\loc.\l.{pong l loc}) o Ping ;;

(* ------- Main class ------- *)
main P @ {p} || ReplyToPing @ locs;;
```

protocol, which will "ping" the nodes and tell the client which nodes respond to the ping. (This simple protocol does not deal with the fact that nodes can fail after responding.)

Fig.1 presents the full EventML specification of the protocol.

### Specification name

The keyword specification marks a specification's name:

```
specification ping_pong
```

### Imports

EventML provides a library file that is a snapshot of Nuprl's library. The types in EventML are a subset of the types in Nuprl. Accordingly, any library function whose type is an EventML type can be used in EventML program. A library function is made visible with an import declaration:

```
import bag-map;;
```

The bag-map function is the map function on bags:

```
bag-map(f;{a,b,...}) = {(f a), (f b),...}
```

### Parameters

Next comes the list of the protocol's parameters. To avoid hardwiring the locations of any participants into the specification, we declare two parameters: p is the location to which clients send their requests; locs is a (non-repeating) bag containing the locations of the nodes to be checked. To execute the protocol

we will instantiate those parameters as real physical machine addresses.[1] A client will identify its location by including that location in the request it sends.

```
parameter p : Loc;;
parameter locs : Loc Bag;;
```

**Messages and directed messages**

A message consists of a *header*, which is a list of tokens, and a *body*, which is a value of a specified type.[2] Our discussions will represent messages as ordered pairs.

A *directed message* is a pair consisting of a location (the addressee) and a message. Directed messages have a special semantics. When a *main* class (see below) produces a bag of directed messages, a messaging system attempts to deliver them—i.e., given the directed message $(loc, msg)$, the messaging system attempts to deliver $msg$ to location $loc$. We reason about the effect of a protocol under assumptions about message delivery. For present purposes, we assume that all messages are eventually delivered at least once, but make no assumption about transit times or the order in which messages are delivered.

The ping-pong protocol uses four kinds of messages, which are declared in a MSGS declaration:

```
MSGS
  input    (``start`` : Loc, base Start)
  internal (``ping``  : Loc, base Ping, send ping)
  internal (``pong``  : Loc, base Pong, send pong)
  output   (``out``   : Loc, send out)
;;
```

The declaration

```
input (``start``  : Loc, base Start)
```

says that messages with header ``*ping*`` (a singleton list of tokens) have bodies of type Loc. It so happens that in this protocol, the body of every kind of message is a location; messages used for different purposes are distinguished by their headers. The keyword input means that ``*start*`` messages are generated by sources outside the protocol; a client sends a ``*start*`` message containing its own location, which will be the return address to which responses from the pinged nodes will eventually be sent. The phrase "base Start" declares Start to be an event class that recognizes the arrival of a ``*start*`` message and observes its body. Such a class is called a *base class*.[3] More precisely, the arrival of a message (``*start*``, $s$) at location $l$, causes an event $e$ to happen at $l$. At event $e$, Start observes the content of that message—which we may formally express in either of two ways:

- $v \in \mathsf{Start}(e)$ if and only if $v = s$
- At $e$, Start returns the singleton bag $\{s\}$

Note that Start cannot observe messages with headers other than ``*start*``.

Ultimately, all EventML programs are defined by applying combinators to base classes, which are the only primitive classes. We assume that any computing system on which we wish to implement EventML provides the means to implement base classes.

In the declarations

[1] As a logical matter, an EventML program may have parameters of any type definable in EventML. To compile an EventML specification, a developer must supply a configuration file that instantiates the parameters. See section 5.

[2] For technical reasons, the Nuprl model represents a message not as a pair but as a triple: the header, the body, and the type.

[3] By convention, the names of event classes—or parameterized event classes—begin with upper case letters.

```
internal (``ping``  : Loc, base Ping, send ping)
internal (``pong``  : Loc, base Pong, send pong)
```

the keyword internal says that ``*ping*`` and ``*pong*`` messages can be sent and/or received only by the ping-pong protocol. (We assume that internal messages cannot be forged.) These declarations introduce the base classes, Ping and Pong, that recognize them. In addition, the keyword send is used to declare functions ping and pong that construct directed messages. For example, if $l$ and $m$ are locations, (ping $l$ $m$) is the directed message for sending to $l$ the message (``*ping*``, $m$).

An output message is generated, but not received by the protocol. Accordingly, the declaration of ``*out*`` messages provides a directed message constructor, but not a base class to recognize them.

**The protocol**

The ping-pong protocol proceeds as follows:

1. The protocol begins when a message of the form (``*start*``, *client*), arrives at location p; *client* is the location that sent it.

2. A supervisory class, P, will then spawn several classes at p. For each $l$ in locs, it spawns the class Handler(*client*,$l$), which will handle communications with node $l$.

3. Handler (*client*, $l$) sends a (``*ping*``,p) message to the node at location $l$ and waits for a response. The message body, p, tells node $l$ where to send its response.

4. On receipt of a (``*ping*``, p) message, the ReplyToPing class at node $l$ sends a (``*pong*``, $l$) message back to p.

5. On receipt of this (``*pong*``, $l$) message, Handler(*client*,$l$) sends an (``*out*``, $l$) message to *client*.

As will be seen, we arrange that a handler class terminates after it has sent an ``*out*`` message.

Note: Nuprl allows subtype definitions, but EventML does not. If subtypes were available in EventML, we could make the message declarations more precise. For example, the body of a ``*ping*`` or ``*pong*`` message cannot be an arbitrary location; it must be one of the locations in locs.

**Class combinators**

Our specification uses the following class combinators:

- Output(f): If f : Loc → Bag($T$), Output(f) is the class that, in response to the first event it sees at location $l$, returns the bag of values (f $l$); it then terminates.

- X || Y: This event class is the parallel composition of classes X and Y. It recognizes events in either X or Y. The parallel combinator is a primitive.

- X >>= Y: This is the *delegation* combinator. If X is an event class and Y is a function that returns event classes, X >>= Y is the event class that, whenever it recognizes an event, acts as follows: For each $v \in$ X($e$), it spawns the class (Y $v$). (Events in this spawned class will occur causally after $e$.) Delegation is primitive.

- f o X: If f is a function that maps values to bags then, very roughly, this class acts as follows: When $v \in$ X($e$), f o X returns (f $v$). (As will be explained, the semantics of this operator are more complex than that.) This *simple composition* combinator is "almost primitive." It is defined in terms of a primitive combinator that is somewhat more expressive but rarely, if ever, used.

- Once(X): This class responds only to the first X-event at any location—and, if $e$ is such an event, $v \in$ (Once(X))($e$) iff $v \in$ X($e$) and there was no $X$-event prior to $e$. Once is a defined combinator.

- X@b: This event class is the restriction of X to the locations in the bag b: $v \in$ (X@b)($e$) iff $e$ occurs at a location in b and $v \in$ X($e$). Operationally, it means "run the program for X at each location in b."

An EventML specification is typically presented in five parts: (1) its name, (2) a list of imported library functions, (3) a parameter list, (4) a list of message declarations, and (5) a list of class declarations. We consider these in turn.

**The "main" class**

The keyword main identifies the event class that compilation of an EventML specification will actually implement (given appropriate instantiations of its parameters). No base class can be a main program. Start, for example, recognizes the arrival of every ``*start*`` message at any node whatsoever; but there is no way to implement that: we cannot install the necessary code on every node that exists (whatever that may mean). On the other hand, Start@{p} can be a main program, since it acts only at location p.

Declaring a class as a main program incurs a proof obligation: one must show that it is, in a technical sense, *programmable*. In particular, a programmable class acts only at some specific, finite collection of nodes. If one defines a main program in an idiomatic way, the proof that it is programmable will be carried out automatically. (The underlying theory: Any class $C$ definable in EventML is *locally programmable*—which means, essentially, that $C@b$ is programmable for any bag $b$ of locations. In addition, all primitive combinators preserve the properties of being programmable and being locally programmable.)

The main program of the ping-pong protocol of the ping-pong program is the parallel composition of the supervisory class P running at location p and ReplyToPing running at all the locations in locs:

```
main P @ {p} || ReplyToPing @ locs;;
```

We will first describe the interactions betwen the Handler classes spawned by P and the nodes, which carry out steps (3)–(5) of the protocol.

**Handler**

Intuitively, Handler is a parameterized class—but, because EventML is a higher-order language we need no special generic or template construct in order to express that. A class parameterized by values of type $T$ is simply a function that inputs values of type $T$ and outputs classes.

The input to Handler is a pair of locations: the client location and the location to ping. A handler is the parallel composition of two other parameterized classes: SendPing, which executes step (3) of the protocol, and ReplyToPong, which executes step (5).

```
class Handler (c,l) = (SendPing (c,l) || ReplyToPong (c,l)) ;;
```

By the definition of the parallel combinator, Handler $(c, l)$ computes everything that either SendPing $(c, l)$ or ReplyToPong $(c, l)$ does.

SendPing $(c, l)$ is in charge of only one task: send a ``*ping*`` message to l.

```
class SendPing (_,loc) = Output(\l.{ping loc l}) ;;
```

The "_" argument is used, as in ML, to make it obvious that the first component of the pair will be ignored. By the definitions of Output and ping given above, an instance of Output$(client, loc)$ running at location $l$ will respond to the first event it sees at $l$ by generating a ``*ping*`` message with body $l$ to location *loc*; it will then terminate. The recipient will interpret $l$ as a return address.

ReplyToPong $(client, loc)$ waits for a ``*pong*`` message from the node at location *loc* and, on receiving one, sends to location *client* an ``*out*`` message with body *loc*. It therefore responds to a *subset* of the events recognized by the base class Pong—not every ``*pong*`` message, but only those sent from *loc*. They are recognized by the fact that the message body consists of the value *loc*. For each $v \in$ Pong$(e)$, we may describe the response of ReplyToPong by saying that it generates a corresponding output by applying the following function to $v$:

```
\l.if l = loc then {out client loc} else {}
```

and then terminates. This function is almost, but not quite, the locally defined function F in the declaration of ReplyToPong:

```
class ReplyToPong (client,loc) =
  let F _ l = if l = loc then {out client loc} else {} in
    Once(F o Pong) ;;
```

The difference is that F takes one extra argument—its first, which, in this case, is ignored.

What is the point of the mysterious extra argument? We assume that any computation processing an event can determine the location at which that event occurs. In general, therefore, a function that is used to transform the value observed by an event class should be able to operate on two arguments—the location of the event and its observed value—even if one or the other of them is ignored.

One can apply simple composition to any number of classes. Given $n$ classes $X_1$, ..., $X_n$, of types $T_1$, ..., $T_n$ respectively, and given a function $F$ of type $\texttt{Loc} \to T_1 \to \cdots \to T_n \to \texttt{Bag}(T)$, one can define a class $C$ by $C = F\ \texttt{o}\ (X_1, \cdots, X_n)$.

Intuitively, $C$ processes an event $e$ as follows. The first argument supplied to $F$ is the location at which $e$ occurs; the successive arguments are, in order, the values observed by the classes $X_i$ at $e$; and $C$ returns the bag that $F$ computes from these inputs. This informal description leaves it unclear what to do if, for some $i$, $e$ is not an $X_i$-event, or what to do if for some $i$, $X_i$ produces a bag with more than one element.

Here is a precise formulation. $C$ produces (observes) the element $v$ of type $T$ iff each class $X_i$ observes an element $v_i$ of type $T_i$ at event $e$ and $v = f\ \texttt{loc}(e)\ v_1\ \cdots\ v_n$. Therefore, a $C$-event must be a $X_i$-event for all $i \in \{1, \ldots, n\}$. If for some $i \in \{1, \ldots, n\}$, $X_i$ does not observe anything at event $e$, then neither does $C$.

**ReplyToPing**

ReplyToPing defines a program that must run at each node that participates in the protocol.

```
class ReplyToPing = (\loc.\l.{pong l loc}) o Ping ;;
```

Once again, it is an instance of the simple composition combinator, and this time the transformation function makes use of the initial location argument (providing the location at which the corresponding program is running). On receiving a (``*ping*``, $l$) message at location $s$, the class (\loc.\l.{pong l loc}) o Ping sends (``*pong*``, $s$) to location $l$.

**Spawning of handlers (delegation to sub-processes)**

The supervisory class P uses the delegation combinator to spawn handlers for each request.

```
class P =
  ((\_.\client.bag-map (\l.(client,l)) locs) o Start) >>= Handler;;
```

We sometimes refer to this combinator as the *bind* combinator because the class type forms a monad and delegation is the bind operator of that monad.

Consider the left-hand side of ">>="; for future reference, call it LHS:

```
(\_.\client.bag-map (\l.(client,l)) locs) o Start
```

When it receives a message (``*start*``, *client*) it produces as output the bag $\{(client, l_1), (client, l_2, \ldots)\}$, where the $l_i$ are the elements of locs. So the effect of LHS >>= Handler is to spawn a class (Handler $(c, l_i)$) for each $i$ each time a (``*start*``, $c$) message arrives. Notice how the types match up: LHS is an event class of type Loc * Loc; the parameterized class Handler is a function mapping values of type Loc * Loc to event classes of type directed message; therefore LHS >>= Handler is an event class of type directed message.

## 3.2 Ping-pong with memory

We now make our ping-pong protocol a bit more interesting by adding some memory to the main process. We introduce a new integer parameter, threshold; instead of sending an (``*out*``, $l$) message to the client whenever node $l$ responds to a pong, we wait until a total of threshold responses have been received, and then notify the client by sending a message (``*out*``, $[l_1; l_2; \ldots; l_{threshold}]$), whose body is the list of all responders. We modify the design of ping-pong by adding one more (parameterized) class, a memory module: Instead of sending an ``*out*`` message directly to a client, ReplyToPong will send an ``*alive*`` alive to an appropriate memory module, which will accumulate responses and send an ``*out*`` message to the client once it has received enough of them.

And we add one more twist. A client who sends multiple ``*start*`` messages will receive multiple ``*out*`` messages in reply and may wish to know what request any ``*out*`` message is replying to. So the client will attach an integer id (we will call it a *request number*) to its ``*start*`` messages, and will receive that request number in the ``*out*`` message that replies. The request numbers need not be globally unique identifiers, so we will also arrange for the supervisory class P to attach a global id (which we will call a *round*) to each request that it receives. The protocol proceeds as follows:

1. P receives a (``*start*``,(*client*,*req_num*)) message from the location *client*.
2. P generates a unique id, *round*, for the request and spawns the following:
   - for each node $l$ in locs, a class Handler($l$, *round*)
   - a memory module (Mem *client req_num round*)
3. Handler($l$,*round*) sends a (``*ping*``,(p,*round*)) message to the node at location $l$ and waits for a reply.
4. On receipt of (``*ping*``,(p,*round*)), the ReplyToPing class at node $l$ sends a (``*pong*``, ($l$, *round*)) message to p. Handler classes respond to ``*pong*`` messages.
5. On receipt of (``*pong*``, ($l$, *round*)), the class Handler($l$, *round*) sends an (``*alive*``, ($l$, *round*)) message to itself (location p). Mem classes respond to ``*alive*`` messages.
6. When (Mem *client req_num round*) has seen ``*alive*`` messages from *threshold* distinct locations, it sends to location *client* an appropriate ``*out*`` message tagged with *req_num*.

Fig. 2 provides the full specification of this protocol. Most of it is a routine adaptation of the ping-pong specification. The novelty lies in the introduction of the event classes PState and Mem that act like state machines. We will describe these in detail.

**Imported library functions**

The specification imports two Nuprl functions.

- deq−member, which checks whether an element belongs to a list

  To apply this to lists of type $T$ we must supply an operation that decides equality for elements of $T$. That operation is a parameter to the membership test; thus, we write (deq−member eq y lst) to compute the value of the boolean "y is a member of list lst, based on the equality test eq."

- length, which computes the length of a list

**Class combinators**

The specification uses the remaining three primitive combinators:

- Prior(X): Event $e$ belongs to Prior(X) if some X-event has occurred at loc($e$) strictly before event $e$; if so, its value is the value returned by X for the most recent such X-event. Once an X-event has occurred at location $l$, all subsequent events at $l$ are Prior(X)-events.

**Figure 2** Ping-pong protocol with memory

```
specification m_ping_pong

(* ------- Imported Nuprl declarations ------- *)
import bag-map deq-member length;;

(* ------- Parameters ------- *)
parameter p : Loc;;
parameter locs : Loc Bag;;
parameter threshold : Int ;;

(* ------- Messages ------- *)
MSGS
  input    (``start`` : Loc * Int, base Start)
  internal (``ping``  : Loc * Int, base Ping,  send ping)
  internal (``pong``  : Loc * Int, base Pong,  send pong)
  internal (``alive`` : Loc * Int, base Alive, send alive)
  output   (``out``   : Loc List * Int, send out)
;;

(* ------- Classes ------- *)
class ReplyToPong p =
  (\slf.\q.if p = q then {alive slf p} else {}) o Pong ;;

class SendPing (loc,round) = Output(\l.{ping loc (l,round)}) ;;
class Handler p = SendPing p || ReplyToPong p ;;

class MemState round =
  let F _ (loc:Loc,r:Int) L =
    if r = round & !(deq-member (op =) loc L) then {loc.L} else {L}
  in F o (Alive,Prior(self)?{[]});;
class Mem client req_num round =
  let F _ L = if length L >= threshold then {out client (L,req_num)} else {}
  in F o (MemState round);;

class Round (client, req_num, round) =
    (Output(\_.locs) >>= \l.Handler (l,round))
  || Once(Mem client req_num round);;

class PState =
  let F @ (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
  in F o (Start,Prior(self)?(\l.{(l,0,0)}));;
class P = PState >>= Round;;

class ReplyToPing = (\loc.\(l,round).{pong l (loc,round)}) o Ping ;;

(* ------- Main class ------- *)
main P @ {p} || ReplyToPing @ locs
```

- X?f: For any class X of type T, and any function $\mathsf{f} : \mathsf{Loc} \to \mathsf{Bag}(\mathsf{T})$, X?f has the following meaning:

$$v \in (\mathsf{X?f})(e) \quad \text{iff} \quad \begin{cases} v \in \mathsf{X}(e) & \text{if } e \text{ is a X-event} \\ v \in \mathsf{f}(\mathtt{loc}(e)) & \text{otherwise} \end{cases}$$

  If (f $l$) is nonempty, then all events at location $l$ are X?f-events.

- self: The underlying semantic model of EventML has powerful operators for defining event classes by recursion, including mutual recursion. However, EventML itself currently provides only a simple recursion scheme, one that has been adequate for all the practical examples we've considered. A recursive definition looks like this:

```
class X = ... Prior(self)?f ...
```

In context, the keyword self refers to the class being defined, in this case X. A definition of this form specifies the value of X at any event $e$ in terms of its value at prior events; and if there is no prior event, in terms of f(loc($e$)). Examples will make this clear.

**P and PState**

Class P uses PState to generate a unique round number for each request, and passes that to Round, which in turn performs step 2 of the protocol. The definition of PState is recursive. Notational convention: Following ML, we use "_" to denote an argument whose value is ignored. In addition, we use "@" to denote an ignored argument of type Loc.

```
class PState =
  let F @ (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
  in F o (Start,Prior(self)?(\l.{(l,0,0)}));;
```

This defines a state machine as follows:

- Start events trigger change of state.
- The state of PState has type (Loc * int * int). For any Start-event $e$, $v \in$ PState($e$) iff $v$ is the state of PState after the occurrence of event $e$.

  The state components represent, respectively: the client whose request has caused the state change, the request number assigned by the client, and the most recent round number generated by PState.
- The initial value of the state at location $l$ is $(l, 0, 0)$.

  The first two components are dummy values.
- The transition function at location $l$ is (F $l$).

  If (``*start*``, (*client*, *req_num*)) arrives in state $(l, r, n)$, the new state is (*client*, *req_num*, $n + 1$).

Here are simple exercises in thinking about such definitions. By definition, PState satisfies the equation:

```
PState =
  let F @ (client,req_num) (_,_,n) = {(client, req_num, n + 1)}
  in F o (Start;Prior(PState)?(\l.{(l,0,0)}));;
```

Because the return value of

```
\l.{(l,0,0)}
```

is always nonempty, every event belongs to the class

```
Prior(PState)?(\l.{(l,0,0)})
```

It follows from this that the PState-events are precisely the Start-events. (The locally defined function F always returns a nonempty result; therefore, for any A and B, the events in F o (A,B) will be those events that are both A-events and B-events.)

Suppose that the arrival of the message (``*start*``, $(c_1, r_1)$) $e_1$ is the first PStart-event occurring at location $l$. Call it event $e_1$. At $e_1$, PState returns

$$\text{F } l \ (c_1, r_1) \ (l, 0, 0) = \{(c_1, r_1, 1)\}$$

Suppose $e_2$ is the next PStart-event occurring at location $l$, which is the arrival of the message (``*start*``, $(c_2, r_2)$). At $e_2$, PState returns

$$\text{F } l \ (c_2, r_2) \ (c_l, r_1, 1) = \{(c_2, r_2, 2)\}$$

The key point is that the argument supplied to F by

```
Prior(PState)?(\l.{(l,0,0)})
```

is the value of the state when the incoming message arrives—which is the value returned as a result of the previous ``*start*`` message (or, if there hasn't been one, $(l, 0, 0)$.

**Mem and MemState**

The state machine PState maintains an internal state and after an input event returns a singleton bag containing its new state. A memory module will maintain an internal state (listing the nodes from which ``*alive*`` messages have been received); it outputs not its state but an ``*out*`` message—and not every change of state will cause an output. A simple way to achieve this is to define two classes: MemState, like PState, simply accumulates a state and makes it visible; Mem observes MemState and generates an output when appropriate.

The class (MemState *round*) accumulates and makes visible the internal state:

```
class MemState round =
  let F _ (loc:Loc,r:Int) L =
    if r = round & !(deq-member (op =) loc L)
    then {loc.L}
    else {L}
  in F o (Alive,Prior(self)?{\l.[]});;
```

An input event to this state machine is the arrival of an ``*alive*`` message. The state is a list of locations, initially empty; it contains the distinct locations from which ``*alive*`` messages have been received for round number round. When a message arrives with body $(loc, r)$ the new state is determined as follows: if the message's round number is round, and *loc* is not yet on the list, prepend *loc* to the state; otherwise, no change. (Because round numbers are globally unique, this class can perform its function without knowing either the client who initiated the request or the request number assigned.)

**Notation:** Some of the formal arguments to the function F are labeled with types: (loc:Loc,r:Int), rather than (loc,r). It is always legal to label arguments in this way; and, in some situations, the type inference algorithm needs the extra help.

**Notation:** Recall that the first argument to deq−member must be an equality operation. In the term "deq−member (op =) loc L" the equality operation is denoted by "(op =)"—meaning that its name is "=" and that, when applied, it will be written as a binary infix operator.

When (Mem client req_num round) sees that the state of (MemState round) has grown to a list of length threshold it signals the client.

```
class Mem client req_num round =
  let F _ L = if length L >= threshold
              then {out client (L,req_num)}
              else {}
  in F o (MemState round);;
```

## 3.3 Leader election in a ring

Many distributed protocols require that a group of nodes choose one of them, on the fly, as a leader. Here is a simple strategy for doing that under the assumptions that:

- the nodes are arranged in a ring (each node knowing its immediate successor)
- each node has a unique integer id

Any node may start an election by sending its own id to its immediate successor (a *proposal*). With one exception, a node that receives a proposal will forward to its successor the greater of the following

**Figure 3** Leader election in a ring

```
specification leader_ring

(* ------- Parameters ------- *)
parameter nodes  : Loc Bag ;;
parameter client : Loc ;;
parameter uid    : Loc → Int ;;

(* ------- Imported Nuprl declarations ------- *)
import imax;;

(* ------- Type functions ------- *)
type Epoch = Int ;;

(* ------- Messages ------- *)
MSGS (* To inform a node of its Epoch and ring successor *)
     input    (``config`` : Epoch * Loc, base Config)

     (* Location of the leader *)
     output   (``leader`` : Epoch * Loc, send send_leader)

     (* Start the leader election *)
     input    (``choose`` : Epoch, base Choose)

     (* Propose a node as the leader of the ring *)
     internal (``propose`` : Epoch * Int, base Propose, send send_propose) ;;

(* ------- Classes ------- *)
let dumEpoch = 0 ;;

class Nbr =
  let F _ (epoch, succ) (epoch', succ') =
    if epoch > epoch'
    then {(epoch, succ)}
    else {(epoch', succ')} in
  F o (Config,Prior(self)?(\l.{(dumEpoch,l)})) ;;

class ProposeReply =
  let F loc (epoch, succ) (epoch', ldr) =
    if epoch = epoch'
    then if ldr = uid loc
         then {send_leader client (epoch, loc)}
         else {send_propose succ (epoch, imax ldr (uid loc))}
    else {}
  in F o (Prior(Nbr),Propose) ;;

class ChooseReply =
  let F loc (epoch, succ) epoch' =
    if epoch = epoch'
    then {send_propose succ (epoch, uid loc)}
    else {}
  in F o (Prior(Nbr),Choose) ;;

(* ------- Main class ------- *)
main (ProposeReply || ChooseReply)  @ nodes
```

two values: the proposal it received, its own id. The exception occurs if (and only if) a node receives in a proposal its own id. In that case, the node stops forwarding messages and declares itself elected. If messages are delivered reliably and no nodes fail, this protocol will always succeed in electing the node with the greatest id.

Fig. 3 presents our specification of a slightly more sophisticated protocol. We add an interface that makes it possible for some external party to reconfigure the ring—e.g., if it believes that some nodes have failed. Informally, we call the intervals between reconfigurations *epochs* (setting aside the vagueness of "between" in a distributed setting). We number the epochs with positive integers—using 0 to mean "no epoch has started at this node."

The inputs to the protocol are of two kinds:

- a ``*config*`` message tells a node to begin a new epoch and stipulates which node is, in the new epoch, its immediate successor in the ring, or
- a ``*choose*`` message contains the number of an epoch, and asks for an election in that epoch.

The outputs of the protocol are ``*leader*`` messages sent to some designated client. The body of a ``*leader*`` message contains an epoch number and the id of the leader elected in that epoch.

Parameters to the protocol are

- nodes : Loc Bag – the nodes from which a leader must be chosen
- client : Loc – the node to be informed of the election results
- uid : Loc → Int – a function assigning a unique id to each member of nodes

Our slightly generalized protocol is still quite simple to describe. A node keeps track of the epoch in which it is currently participating and ignores all ``*propose*`` or ``*choose*`` messages labeled with other epochs. If it receives a ``*config*`` message for an epoch numbered higher than its current epoch, it switches to the new epoch, and otherwise ignores it. A node reacts to all non-ignored ``*propose*`` and ``*choose*`` messages as in the original protocol.

The delicate part lies in formulating the invariants preserved by the protocol and the conditions under which it succeeds. What if reconfiguration occurs while an election is going on? What if ``*config*`` messages arrive out of order—requesting epoch 4 and later requesting epoch 3? What if ``*config*`` messages partition the nodes into two disjoint rings? Those questions are not the subject of this note.

**Nbr, the state of a node**

Informally, the state of any node is a pair $(epoch, succ) : \mathsf{Int} * \mathsf{Loc}$, where *epoch* is the number of its current epoch and *succ* is the location of its current successor. This state changes only in response to ``*config*`` messages. We capture this behavior in the class Nbr, which defines a state machine as follows:

- At location $l$, its initial state is $(0, l)$; essentially, these are both dummy values.
- Input events are the arrivals of ``*config*`` events, which are recognized by the base class Config.
- The state transition in response to the input $(epoch', succ')$ is: if $epoch' > epoch$, then change to $(epoch', succ')$; otherwise, no change.

We use the state machine idiom described in section 3.2.

```
let dumEpoch = 0 ;;

class Nbr =
  let F _ (epoch, succ) (epoch', succ') =
    if epoch > epoch'
    then {(epoch, succ)}
    else {(epoch', succ')} in
  F o (Config, Prior(self)?(\l.{(dumEpoch, l)})) ;;
```

**Factoring the main program.**

We factor the behavior of the protocol into two classes, one triggered by ``*propose*`` messages and one triggered by ``*choose*`` messages. We define

```
main (ProposeReply || ChooseReply) @ nodes
```

and will define ProposeReply and ChooseReply in terms of Nbr.

**ProposeReply.**

The response to a proposal is as described informally: send a ``*leader*`` message if you receive your own id; otherwise, propose the max of the proposal received and your own id.

```
class ProposeReply =
  let F loc (epoch, succ) (epoch', ldr) =
    if epoch = epoch'
    then if ldr = uid loc
         then {send_leader client (epoch, loc)}
         else {send_propose succ (epoch, imax ldr (uid loc))}
    else {}
  in F o (Prior(Nbr),Propose) ;;
```

The functions send_leader and send_propose are the directed message constructors introduced in the declarations of ``*leader*`` and ``*propose*`` messages.

Since Nbr changes only in response to ``*config*`` messages, the state of Nbr is the same both before and after a ``*propose*`` message arrives. So why couldn't we simplify this definition by replacing the expression "F o (Prior(Nbr),Propose)" with "F o (Nbr,Propose)"?

The reason is that Nbr can only observe Config events, whereas Prior(Nbr) can observe any event $e$ such that an Config event has previously occurred at $\texttt{loc}(e)$. This use of Prior (...) is a basic idiom of EventML programming.

Note: If $e$ is a Propose event at location *loc*, and no Config event has yet occurred at *loc*, then $e$ is not a Prior(Nbr) event, and therefore is not a ProposeReply event.

**ChooseReply**

When ChooseReply receives a ``*choose*`` instruction for the epoch on which it is currently working, it initiates an election by sending an appropriate ``*propose*`` message.

```
class ChooseReply =
  let F loc (epoch, succ) epoch' =
    if epoch = epoch'
    then {send_propose succ (epoch, uid loc)}
    else {}
  in F o (Prior(Nbr),Choose) ;;
```

This applies the same "Prior (...) " idiom used in the definition of ProposeReply.

## 3.4 Interlude: State machines

The Nuprl library defines combinators that package up idioms for defining various kinds of state machines.

**Accum−class**

The simplest state machine combinator is Accum−class. If

- A is a type – representing input values

- B is a type – representing values of the class's internal state
- X is an event class of type A – recognizing input events
- init_state : Loc → B Bag – assigning initial states to locations
- f : A → B → B – the transition function

then we may interpret (Accum−class f init_state X) as a class of type B that acts like a state machine with the given initial values and transition function. More precisely, if init_state assigns a singleton bag to every location, then

- The events this class recognizes are the X-events.
- To every X-event $e$ it assigns a singleton bag; the element of that bag is the state of that state machine after responding to $e$.

Note that the Prior combinator will allow us to observe the state when the input arrives and before it is processed.

The Accum−class combinator is defined as follows:

```
class Accum−class f init_state X =
  (\_.f) o (X, Prior(self)?init_state);;
```

Thus, if we declare

```
Class Y = Accum−class f init_state X;;
```

we know that Y satisfies the equation

```
Y = (\_.f) o (X, Prior(Y)?init_state)
```

**Threshold−Combinator.**

Accum−class defines a state machine that maintains an internal state and allows us to observe the state (by making appropriate use of Prior(_) and _?_) but does not, in response to state changes, issue any other outputs.

Threshold−Combinator provides a useful way to define a state machine that issues outputs in response to inputs. The setting: we're given an event class X of type A that recognizes input events and want to define a state machine that maintains an internal state of some type S and, computes outputs of type B based on the inputs and its current state.

So we will have to supply an output function in addition to a state transition function. We do not supply them directly as inputs to Threshold−Combinator; rather, we supply inputs from which they can be determined in a slightly indirect way:

```
class Threshold−Combinator R X init_state accum f =
  let state_function v state =
     if R v state then accum v state else state in
  let output_function loc v state =
     if R v state then f loc v state else {} in
  let CurrentState = Accum−class state_function init_state X in
  let PriorState = Prior(CurrentState)?init_state in
    output_function o (X, PriorState)
```

The locally defined operations have the following meanings:

**Figure 4** Simple 2/3 consensus - header

```
specification RSC

(* ------- Parameters ------- *)
(* consensus on commands of aribtrary type Cmd with equality decider *)
parameter Cmd, cmdeq : Type * Cmd Deq;;
parameter flrs     : Int      ;; (* max number of failures                  *)
parameter locs     : Loc Bag  ;; (* locations of (3 * flrs + 1)   replicas  *)
parameter clients  : Loc Bag  ;; (* locations of the clients to be notified *)

(* ------- Imported Nuprl declarations ------- *)
import length poss-maj list-diff deq-member from-upto Threshold-Combinator   ;;

(* ------- Type definitions ------- *)
type Inning = Int ;;
type CmdNum = Int ;;
type RoundNum = CmdNum * Inning ;;
type Proposal = CmdNum * Cmd ;;
type Vote = (RoundNum * Cmd) * Loc ;;

(* ------- Messages ------- *)
MSGS
  internal (``sc vote``     : Vote, base Vote, broadcast voteMSGSto)
  internal (``sc retry``    : RoundNum * Cmd, base Retry, send retryMSGto)
  internal (``sc decided``  : Proposal, base Decided, send dcdMSGto)
  output   (``sc notify``   : Proposal, broadcast notifyMSGS)
  input    (``sc propose``  : Proposal, base Propose);;
```

- CurrentState is a state machine that accumulates, and allows us to observe, its state: X recognizes inputs; init_state defines initial states; and the transition function state_function is a *variant* of the parameter accum.
- PriorState, according to our standard idiom, remembers the most recent value of CurrentState.
- output_function, a *variant* of the parameter f, computes an output from the input and the value of PriorState.

To understand the meanings of state_function and output_function, note the expected types of the parameters:

- R : A → S → Bool
- accum: A →S → S
- f: Id → A → S → B Bag

R is a decidable relation between input values and values of the state used to filter inputs. When input v arrives in state s, then: if R holds, the state transition and output are given by accum and f; otherwise, there is no state change and no output.

## 3.5 Two-thirds consensus

Consider the following problem: A system has been replicated for fault tolerance. It responds to commands issued to any of the replicas, which must come to consensus on the order in which those commands are to be performed, so that all copies process commands in the same order. Replicas may fail. We assume that all failures are crash failures: that is, a failed replica ceases all communication with its surroundings. The two-thirds consensus protocol is a simple protocol for coming to consensus, in a manner that

**Figure 5** Simple 2/3 consensus - body

```
(* ------- 2/3 majority consensus, aka "simple consensus" ------- *)
let roundout loc (((n,i),c),_:Loc) (cmds,_:Loc List) =
  if length cmds = 2 * flrs
  then let (k,x) = poss-maj cmdeq (c.cmds) c in
       if k = 2 * flrs + 1
       then { dcdMSGto loc (n, x) }
       else { retryMSGto loc ((n,i+1), x) }
  else {} ;;

let init x @ = {x} ;;
let thr_out @ x _ = {x};;

let newvote (ni:RoundNum) ((ni',c),sender:Loc) (_:Cmd List,locs) =
  ni = ni' & !(deq-member (op =) sender locs);;
let addvote ((_:RoundNum,c),sender) (cmds,locs) = (c.cmds, sender.locs);;
class Quorum ni =
  Threshold-Combinator (newvote ni) Vote (init (nil,nil)) addvote roundout;;

class Round (ni,c) = Output(\loc.voteMSGSto locs ((ni,c),loc))
                   || Once(Quorum ni) ;;

let vote2retry @ (x,loc) = {x};;
class RoundInfo = Retry || (vote2retry o Vote);;

let round_increase (n:CmdNum) ((m,i),c) round = n = m & round < i;;
let incround ((_,i),_) s = i;;
class NewRounds n =
  Threshold-Combinator (round_increase n) RoundInfo (init 0) incround thr_out ;;

let decision n @ (m,c) = if m = n then notifyMSGS clients (m,c) else {};;
class Notify n = Once((decision n) o Decided);;

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

let onnewpropose (n,_) (max, missing) =
  if n > max
  then (n, missing ++ (from-upto (max + 1) n))
  else (max, list-diff (op =) missing [n]);;

let vote2prop @ (((n,i),c),@) = {(n,c)} ;;
class Proposal = Propose || (vote2prop o Vote);;

let new_proposal (n,c) (max,missing) = n > max or deq-member (op =) n missing;;
class NewVoters =
  Threshold-Combinator new_proposal Proposal (init (0,nil)) onnewpropose thr_out ;;

class Replica = NewVoters >>= Voter;;

main Replica @ locs ;;
```

tolerates $n$ failures, by using $3n+1$ replicas. We will describe how the protocol works, without explaining why it works.

Input events communicate *proposals*, which consist of (integer,command) pairs: $(n, c)$ proposes that command $c$ be the $n^{th}$ one performed. The arrival of a proposal is an event recognized by the base class Propose. The header of such a message is ``*propose*``. The protocol is intended to obtain consensus, for each $n$, on which command will be the $n^{th}$ to be performed, and to broadcast those decisions (which are also integer/command pairs) to a list of clients. Decision messages will have the header ``*notify*``.

Figures 4 and 5 present the full specification.

**Comments on figure 4**

The parameters are

- Cmd: the type of commands.
- flrs : the max number of failures to be tolerated
- locs: the locations of the $3 * \text{flrs} + 1$ replicated deciders
- clients : the locations of the clients to be notified of decisions

We make no assumptions about or constraints on who submits inputs.

The declaration of the Cmd parameter also introduces a parameter that for an equality operator:

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

When we instantiate the type Cmd, we must also instantiate cmdeq with an operation that decides equality for members of that type. The keyword Deq denotes a type constructor: (Cmd Deq) is the type of all equality deciders for Cmd. We need cmdeq because the operation deq−member, which decides membership in a list, requires as one of its arguments an operation that decides equality for members of the list (section 3.2).

Officially, the message headers are the token lists ``*sc vote*``, ``*sc retry*``, etc. Informally, we will denote these by ``*vote*``, ``*retry*``, etc.

The declarations of ``*vote*`` and ``*notify*`` messages introduce constructors for broadcasting directed messages, via the keyword broadcast:

```
internal (``sc vote``: Vote, base Vote, broadcast voteMSGSto)
```

If $v$ is a Vote and the $l_i$ are locations,

$$\text{voteMSGSto}\ [l_1, \ldots, l_n]\ v = \{(l_1, (``vote``, v)), \ldots, (l_n(``vote``, v))\}$$

That is, it constructs a bag of directed messages that send a $(``vote``, c)$ message to each location $l_i$.

**Class combinators**

The specification uses one new defined combinator:

- X until Y: $v \in (\text{X until Y})(e)$ iff $v \in \text{X}(e)$ and no Y-event has previously occurred at $\texttt{loc}(e)$. That is, at any location $l$, the class (X until Y) acts exactly like X until a Y-event occurs at $l$, after which it falls silent.

**The top level**

Replica is the event class characterizing the actions of a decider. The main program

```
main Replica @ locs
```

installs a decider at each location in locs.

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

NewVoters will respond (only) to the first proposal $(n, c)$ it sees for command $n$; and it responds by spawning (Voter $(n, c)$), which will negotiate with voters spawned by other Replicas about which proposal for command $n$ to accept.

We define consensus on proposal $(n, c)$ to mean that 2/3 (plus one) of the replicas vote for it. On any particular vote, that degree of consensus cannot always be guaranteed—so we allow do-over votes, for which we adopt the following terminology. Successive votes for each command number are assigned consecutive integers, starting with 0, called *innings*; the pair (command_number, inning) is called the voting *round*.

Votes are of type Vote, and each contains:

- the round in which the vote is cast
- a command being voted for in that round
- the voter's location (to ensure that no replica gets more than one vote)

So a Voter is a parallel composition of three classes:

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

where:

- Round$((n, 0), c)$ will, at any location, conduct the voting for round $(n, 0)$, and will cast its vote in that round for command $c$.
- Notify $n$ will recognize when agreement has been reached as to what should be the $n^{th}$ command and broadcast the result.
- NewRounds $n$ >>= Round will determine when it is time to begin a new inning of voting for the $n^{th}$ command and spawn a class to conduct the voting; because of the "until (Notify n)" this third component will terminate at any location once a (Notify $n$)-event occurs there.

### Specifying state machines

We introduce some convenient notation for specifying the init_state and f parameters of Threshold−Combinator (section 3.4):

```
let init x @ = {x} ;;          (* -- for specifying "init_state" *)
let thr_out @ x _ = {x} ;;     (* -- for specifying "f" *)
```

Instantiating init_state with (init $v$) assigns the initial state $v$ to every location. Instantiating f with thr_out defines a state machine that acts purely as a filter: the value of an output is the value of the input that caused it.

### NewVoters

As noted, NewVoters is a filter: Its internal state keeps track of the proposals received at each location; for each $n$, it recognizes the first proposal of form $(n, c)$ that it sees, and transmits that proposal as its output. The proposals it considers may come from an external input or from a vote. The class Proposal recognizes these input events:

```
let vote2prop @ (((n,i),c),@) = {(n,c)} ;;
class Proposal = Propose || (vote2prop o Vote);;
```

Propose is the base class that recognizes input events (header ``*propose*``); Vote is the base class that recognizes the arrival of votes (header ``*vote*``).

```
class NewVoters =
  Threshold-Combinator new_proposal
                       Proposal
                       (init (0,nil))
                       onnewpropose
                       thr_out ;;
```

Here is the correspondence between (some of) the actual formal parameters parameters and types of the definition:

- X $\sim$ Proposal, which recognizes inputs
- A $\sim$ the input type: votes
- B $\sim$ the output type: votes
- f $\sim$ thr_out, because this class is a filter
- S $\sim$ the type of the internal state: Int $*$ (Int List), initial value $(0, nil)$

Intuitively, the internal state of NewVoters is a pair $(max, missing)$ : Int $*$ (Int List), where: $max$ is the greatest natural number for which a proposal has thus far been seen (at a particular location); $missing$ is the list of of all integers less than $max$ for which no proposal has been received.

The remaining parameters:

- R $\sim$ new_proposal, which recognizes the proposals that should cause outputs
- accum $\sim$ onnewpropose

NewVoter can change state and generate an output on input $(n, c)$ if it has never before seen a proposal for the $n^{th}$ command. Accordingly,

```
let new_proposal (n,c) (max,missing) =
  n > max or deq-member (op =) n missing;;
```

When it sees a new proposal NewVoters updates $(max, missing)$ so as to maintain the invariant that the state keeps track of the highest number it has seen and the list of all the lower numbers it has not:

```
let onnewpropose (n,_) (max, missing) =
  if n > max
  then (n, missing ++ (from-upto (max + 1) n))
  else (max, list-diff (op =) missing [n]) ;;
```

where ++ is the append operator and the imported Nuprl operations from-upto and list-diff have the following meanings:

```
from-upto i j = [i; i+1; i+2; ...; j-1]

list-diff (op =) [a;b;...] [n] =
    the result of deleting any occurrences of n from [a;b;...]
```

**Notify n**

A ``*decided*`` message signals that a consensus has been reached; its data is the proposal agreed on. It is an internal message. (Notify $n$) waits for a ``*decided*`` message about the $n^{th}$ command and responds by broadcasting that decision to all clients. It terminates after it has seen one such message. (The base class Decided recognizes ``*decided*`` messages.)

```
let decision n @ (m,c) = if m = n
                         then notifyMSGS clients (m,c)
                         else {};;
class Notify n = Once((decision n) o Decided);;
```

**Round** $((n, i), c)$

Round $((n, i), c)$, running at location *loc*, does three things: When launched, it broadcasts a vote from *loc* for command $c$ in round $(n, i)$; in addition, it keeps a tally of votes received at $l$; it uses that tally to determine *either* that consensus has been reached (in which case it sends to itself a ``*decided*`` message) *or* that consensus might not be possible in inning $i$ (in which case it sends to itself a suitable ``*retry*`` message).

```
class Round (ni,c) = Output(\loc.voteMSGSto locs ((ni,c),loc))
                   || Once(Quorum ni) ;;
```

The class (Quorum $ni$) defines the state machine that keeps a tally for round $ni$, and sends an appropriate message when a quorum is reached (or judged possibly unattainable):

```
class Quorum ni =
  Threshold-Combinator (newvote ni)
                       Vote
                       (init (nil,nil))
                       addvote
                       roundout;;
```

Again we note the correspondence between the formal and actual parameters to Threshold−Combinator:

- X ∼ Vote, which recognizes inputs
- A ∼ the input type: votes
- B ∼ the output type: directed messages
- S ∼ the the type of the internal state: (Cmd List) ∗ (Loc List), initial value $(nil, nil)$

Intuitively, the state consists of a pair $(cmds, locs)$, where: *cmds* is the list of commands that have been proposed in inning $i$ as the $n^{th}$ command; *locs* is the (non-repeating) list of the locations that sent those commands. We keep a list of senders so that, if a vote from any sender is delivered multiple times, it will only be counted once.

- R ∼ newvote

State transitions occur, and outputs may be generated, when a new vote arrives—i.e., a vote in round $ni$, sent from a location not previously heard from:

```
let newvote (ni:RoundNum) ((ni',c),sender:Loc) (_:Cmd List,locs) =
   ni = ni' & !(deq-member (op =) sender locs);;
```

- accum $\sim$ addvote

When a new vote arrives, we update the state by prepending to its components the command it votes for and the location of its sender:

```
let addvote ((_:RoundNum,c),sender) (cmds,locs) =
        (c.cmds, sender.locs);;
```

- f $\sim$ roundout

Each new vote causes a state transition, but need not cause an output. Round $((n, i), c)$ sends an output message once it has received votes from 2 flrs $+1$ distinct locations. If all of them are votes for the same command $d$, it sends itself the ``*decided*`` message with the proposal $(n, d)$. If not, then it is possible that on this round no proposal will ever receive 2 flrs $+1$ votes; so it sends itself a ``*retry*`` message to trigger initiation of inning $i + 1$. (Once it has sent the ``*retry*`` message it will ignore any votes it subsequently receives in round $(n, i)$, even if they would result in some proposal's receiving 2 flrs $+1$.) The data of a ``*retry*`` message consists of the new round to be initiated and, in addition, the name of a command (which will be proposed in this new inning). The definition of roundout attempts to choose that command in a reasonable way: poss-maj implements the Boyer-Moore majority algorithm. So, if the votes are not unanimous, but some command receives a majority, that majority-receiving command will be proposed in the ``*retry*`` message.

```
let roundout loc (((n,i),c),_:Loc) (cmds,_:Loc List) =
  if length cmds = 2 * flrs
  then let (k,x) = poss-maj cmdeq (c.cmds) c in
         if k = 2 * flrs + 1
         then { dcdMSGto loc (n, x) }
         else { retryMSGto loc ((n,i+1), x) }
  else {} ;;
```

**NewRounds n**

(NewRounds $n$) is a filter. It keeps track, at any location, of the greatest $i$ for which it has participated in a round of the form $(n, i)$; if it receives a ``*retry*`` message with data $((n, j), c)$, or a vote $(((n, j), c), loc)$, and $j > i$, it remembers $j$ and outputs the data $((n, j), c)$.

```
class NewRounds n =
  Threshold-Combinator (round_increase n)
                       RoundInfo
                       (init 0)
                       incround
                       thr_out ;;
```

So, we have:

- X $\sim$ RoundInfo, which recognizes inputs (which may be votes or retries)
- A $\sim$ the input type: Round $*$ Cmd

- B $\sim$ the output type: Round $*$ Cmd
- S $\sim$ the internal state: Int, initial value 0
- f $\sim$ thr_out

```
let vote2retry @ (x,loc) = {x};;
class RoundInfo = Retry || (vote2retry o Vote);;
```

- R $\sim$ round_increase

As noted, round_increase $n$ identifies a retry or vote for command $n$ that applies to an inning greater than the current one.

```
let round_increase (n:CmdNum) ((m,i),c) round =
    (n = m & round < i);;
```

- accum $\sim$ incround

When a new, higher inning, is initiated, the state is reset to that inning number:

```
let incround ((_,i),_) s = i;;
```

# 4 Definitions of combinators

**General simple composition**

Section 3.1 introduces the simple composition combinator. Given $n$ classes $X_1$, ..., $X_n$, of types $T_1$, ..., $T_n$ respectively, and given a function $F$ of type Loc $\rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow$ Bag$(T)$, one can define the class $F$ o $(X_1, \cdots, X_n)$. This combinator is defined in terms of one of the Logic of Events' primitive combinators. Given $n$ classes $X_1$, ..., $X_n$, of types $T_1$, ..., $T_n$ respectively, and given a function $F$ of type Loc $\rightarrow$ Bag$(T_1) \rightarrow \cdots \rightarrow$ Bag$(T_n) \rightarrow$ Bag$(T)$, the class $F$ O $(X_1; \cdots; X_n)$ is one of the Logic of Events' primitive combinator. The class $F$ o $(X_1, \cdots, X_n)$ is defined as:

$$(\lambda loc.\ \lambda b_1 \ldots . \lambda b_n.\ \bigcup_{x_1 \in b_1} \cdots \bigcup_{x_n \in b_n} F\ loc\ x_1\ \cdots\ x_n)\ \mathtt{O}\ (X_1; \cdots; X_n)$$

**Until**

The binary infix operator until can then be defined in terms of this more general simple composition combinator as follows:

```
import bag-null;;

class until X Y =
  let F loc b1 b2 = if bag-null b2 then b1 else {}
  in F O (X, Prior(Y)) ;;

infix until;;
```

The bag−null function is a function that returns true iff its argument is the empty bag. Note that using a infix declaration, one can declare infix operator in EventML.

**Once**

The Once operator can be defined in terms of the until operator as follows:

```
class Once X = (X until X) ;;
```

**Output**

The Output operator can be defined in terms of the Once operator as follows:

```
class Output b = Once(b o ());;
```

**The "at" combinator**

The binary infix operator @ can be defined in terms of the more general simple combinator as follows:

```
import bag-deq-member ;;
class @ X locs =
  let F loc x = if bag-deq-member (op =) loc locs then {x} else {}
  in F o X ;;
infix @ ;;
```

(Note that this code is not valid EventML code because @ is not a valid identifier.)

**Parallel combination**

The parallel combinator can be defined in terms of the more general simple combinator as follows:

```
class || X Y = (\loc.\b1.\b2.b1++b2) o (X,Y) ;;
infix || ;;
```

(Note that this code is not valid EventML code because || is not a valid identifier.)

**Accum-class Threshold-Combinator**

The two combinators Accum−class and Threshold−Combinator are defined in section 3.4.

# 5 Configuration files

Parameters to our specifications are of two kinds. Some are "abstract"—e.g., the integer parameters threshold (see section 3.2) and flrs (see section 3.5). We can instantiate these by providing a Nuprl term of type integer. Others are "real world"—e.g., the parameter client of type location. Their meanings are specific to a particular installation of EventML: the messaging system determines what must be supplied to instantiate a location parameter. Our prototype assumes that messaging is by TCP/IP, and a location is a pair consisting of an IP address and a port.[4]

The parameter declarations

```
parameter nodes  : Loc Bag ;;
parameter client : Loc ;;
parameter uid    : Loc → Int ;;
```

[4]Note that TCP/IP provides stronger guarantees—namely, FIFO delivery—than our examples have assumed.

illustrate the open-ended nature of real world parameters.

Suppose that we supply an (IP address, port) pair for client and a list of such pairs for nodes.[5] How do we instantiate uid? Knowing the locations, we could simply define a function that assigns integers to them. If we wanted a more flexible implementation, we might want to base uid on the MAC address of a node's network card; in that case the configuration file would provide some reference to a piece of code that does the computation. For now, the only primitive real-world type that we allow is Loc. All other parameter types must be intepretable from Loc and abstract types.

Here is what a configuration file looks like:

```
%locations
 n1: 192.168.0.12 19777
 n2: 192.168.0.13 19778
 n3: 192.168.0.14 19779

%parameters
 nodes:  {LOC(n1);LOC(n2);LOC(n3)}
 client: LOC(client)
 uid:    \l.if l = LOC(n1) then 1 else if l = LOC(n2) then 2 else 3

%messages
 n1: (``config'', Int * Loc, (1, LOC(n2)))
 n2: (``config'', Int * Loc, (1, LOC(n3)))
 n3: (``config'', Int * Loc, (1, LOC(n1)))
 n2: (``choose'', Int, 1)
```

This is an example of a configuration file for the leader election in a ring protocol presented in section 3.3. A configuration file is divided into three parts: the locations part declares the machines on which one wishes to install the specified protocol (n1 is a location name which is specified by the IP address 192.168.0.12 and the port number 1977); the parameters part instantiates the parameters declared in the given specification (the leader election in a ring specification presented in section 3.3 declares three parameters: nodes, client, and uid); the messages part declares a bag of messages initially in transit. One has to declare at least one message in transit because EventML allows on to define reactive agents that can only react on receipt of messages. Therefore nothing happens as long as no message is received.

# References


[ABC+06] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using nuprl. *J. Applied Logic*, 4(4):428–469, 2006.

[BC08] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[BCG10] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical report, Cornell University, 2010.

[BCH+00] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *In DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160. IEEE Computer Society Press, 2000.


[5] Computationally, a bag is just a list in which we ignore the order.

[Bic09] Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *LNCS*, pages 140–155. Springer, 2009.

[BKR01] Mark Bickford, Christoph Kreitz, and Robbert Van Renesse. Formally verifying hybrid protocols with the nuprl logical programming environment. Technical report, Cornell University, 2001.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CHP84] Guy Cousineau, Gérard Huet, and Larry Paulson. *The ML handbook*, 1984.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Hay98] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, Department of Computer Science, 1998. Technical Report TR98-1662.

[KHH98] Christoph Kreitz, Mark Hayden, and Jason Hickey. A proof environment for the development of group communication systems. In *Automated Deduction - CADE-15, 15th Int'l Conf. on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 1998.

[KR11] Christoph Kreitz and Vincent Rahli. *Introduction to Classic ML*, 2011.

[Kre02] Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide.* Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Ler00] Xavier Leroy. *The Objective Caml system release 3.00.* Institut National de Recherche en Informatique et en Automatique, 2000.

[LKvR+99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth P. Birman, and Robert L. Constable. Building reliable, high-performance communication systems from components. In *SOSP*, pages 80–92, 1999.

[Ren11] Robbert Van Renesse. Paxos made moderately complex. 2011.

[Win88] Glynn Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, volume 354 of *LNCS*, pages 364–397. Springer, 1988.

# Index