Interfacing with Proof Assistants for Domain Specific Programming Using EventML

Vincent Rahli

Cornell University

Abstract

This paper presents a paradigm for using proof assistants in the programming process. We demonstrate how the programming language EventML provides a useful interface to proof assistants throughout code development. Enhancing the use of proof assistants in programming will make them more valuable and accessible to a large community. We designed EventML specifically to cooperate with proof assistants at every stage of program creation. It will help programmers ensure correctness, document the code, and support modifications and improvements. Cooperation is especially effective when the proof assistant and the programming language share the same type system and when the proof assistant can generate executable code for the programming language, as is natural for proof assistants using constructive type theories.

1 Introduction

Program development and verification using formal tools. Developing and maintaining software systems remains error prone. One can only assume that unverified programs are wrong which often turns out to be true. Unfortunately, the task of verifying the correctness of a program using formal tools such as proof assistants remains costly. Proof assistants face the barrier that they are an advanced technology requiring specialized education. Moreover, some of them rely on such expressive theories that full automation is impossible. Users then have to learn how to communicate with these tools to create proofs, which can be a daunting task, especially for beginners. A wide range of approaches have been proposed to facilitate the communication between humans and proof assistants, such as providing user friendly interfaces [45, 4, 2], but also providing *domain specific* frameworks and proof techniques.

This paper claims that one of the most promising ways to make proof assistants more useful and widely adopted is to connect them more completely into the programming process. Much work has been done towards bridging the gap between the programming and verification tasks. To name a few, Rushby [42] and Pike [39] proposed a partially verified mechanism to derive synchronous time-triggered implementations from functional specifications; Lynch et al. [21, 32, 25] developed IOA, TIOA, and Tempo, which are frameworks providing support for verification and synthesis of asynchronous distributed systems specified as I/O automata; The Ensemble toolkit was verified and optimized using the Nuprl proof assistant [30] via an embedding of a subset of OCaml in Nuprl using the close connection between OCaml and Nuprl's underlying programming language; PL/CV [19] mixes commands and assertions in a single programming language/logic which enables formal program reasoning directly in the code. The Spec# programming system [6, 5] also features specification constructs (such as pre- and postconditions) which are checked both dynamically and statically; F* [44] is a dependently-typed programming language which aims, among other things, at verifying protocols using types (such as refinement types) expressive enough to formalize and reason about security properties. Epigram [34, 33] is a functional programming language that also features dependent types in order to "reduce certification to type checking" [3]. The TRELLYS [43] project also advocates the use of dependently-typed programming languages as lightweight verification tools. Bickford et al. [13] synthesize correct-by-construction distributed programs from high-level specification written in a theory of events.

However, even when using such frameworks, it can still be hard to obtain "full" correctness of the code either (1) because of the gap between the languages used on the programming and verification sides, or (2) because the language used on the verification side is not abstract enough, or (3) because the underlying logic is either too weak or does not provide enough support/automation to formally verify all necessary properties.

Proof assistants that implement constructive type theories such as Agda [20, 15], Coq [9, 1], Isabelle [8, 7], or Nuprl [18, 26, 2] are especially well-suited to bridge the gap between the programming and verification tasks because they allow one to synthesize code that provably satisfies high-level specifications. This mechanism is called *program extraction*, and the obtained code is said to be *correct-by-construction*. For example, in Nuprl, proofs have computational content, namely programs which serve as evidence for the truth of statements, and which can automatically be extracted, thereby creating programs that meet specifications given by theorems. Constructive proof assistants that implement this mechanism provide a powerful framework for developing programming languages based on "executable" specifications. Such a high-level programming language featuring a (verified [29, 17]) compiler based on program extraction is a good candidate for the "programming language of the future". An important issue while developing such a programming language is to find the "right" level of abstraction in order to facilitate program reasoning.

A paradigm for verified programming. The method of the paradigm proposed in this paper is to provide a high-level programming language which gives precedence to the programming task, but also provides an entry point to formal methods by allowing programmers to cooperate with a proof assistant in order to structure arguments for correctness. To reach a wide audience, this programming language should be based on a well-established, widely used programming language family such as ML, which has the advantage of being the metalanguage of several proof assistants such as the ones mentioned above. We call our language EventML because it is an ML dialect targeted to develop networks of distributed agents that react to *events*, where an event is an abstract object corresponding to a point in space/time that has information associated to it (more concretely, an event can be seen as the receipt of a message by an agent). Using EventML, programmers specify systems at a high-level of abstraction, and EventML automatically synthesizes code from such specifications. (EventML's code synthesizer is highly reliable because it mimics Nuprl's code synthesizer which generates correct-by-construction code that *provably* satisfies the given specification.)

Programmers can gain confidence in their specifications by running the synthesized code. However, to fully trust that code, they eventually have to interact with, or "dock" to, the Nuprl proof assistant where they can load their specifications, prove their correctness, and synthesize code using the extraction mechanism. Nuprl is a Logical Programming Environment featuring a large database of definitions, facts, and extracts. Therefore, programmers can also benefit from docking to Nuprl by downloading verified specifications stored in its database.

Formulating adequate specifications is itself a difficult and error prone task. It is inextricably linked to any effort to start writing code, from the choice of data types, choice of function and procedure definitions, organization of the code into modules/classes, etc. Discovering an error in a specification is no less disastrous than discovering an error in a program, and might mean the loss of a few days worth of proving some of its properties. To partially overcome that inefficiency, EventML provides a range of program annotations, such as invariants, that tie together a specification with the correctness argument that its writer has in mind. These annotations can be loaded in Nuprl along with specifications, and we wrote tactics that attempt to prove them automatically. Programmers can then gain extra confidence in their specifications if Nuprl successfully proves these statements. Another advantage of such logical statements is that they serve the purpose of documentation. Using this cooperation between the EventML programming tool and the Nuprl formal method tool, programmers can gradually harden their specifications (e.g., by fixing bugs or by making them more general or more specific as needed).

Finally, we believe this paradigm is a step towards balancing the tendencies in the programming language and formal methods communities: on the programming side, in order to get reliable code, one has to use formal methods; and on the formal method side, the machinery developed to model systems has to be abstract enough to make the verification task tractable but not too abstract in order to be able to synthesize code and, moreover, issues such as efficiency have to be taken into account.

EventML's features. The EventML language[1] allows one to quickly specify a protocol in the Emacs text editor, make assertions, type check the specification, synthesize code, and simulate or run that code in a real distributed environment.

EventML's language is higher-order, strongly typed, and features a type inferencer based on ML's. EventML's type inferencer is constraint-based [36, 40], i.e., it can be divided into two non-interleaving phases: first, given a piece of code, constraints (e.g., type equality constraints) are generated; then, a constraint solver is used to solve the generated constraints and therefore check the typablity of the piece of code. Among other things, a constraint-based type inferencer allows us to do type error slicing [24, 41], i.e., to report type errors as portions of code (called slices) which, in EventML, can be highlighted directly in the code using our Emacs interface.

EventML also provides a simulator, as well as a message system (implemented on top of TCP) to run the synthesized processes in a real distributed environment. EventML's code synthesizer generates distributed processes [11] coded in the syntax of Nuprl's term language—an extension of the untyped λ-calculus. Therefore, to simulate or run these processes, EventML features several Nuprl term evaluators that all respect Nuprl's semantics (e.g., evaluation in Nuprl is lazy).

Contributions. The contributions of this paper are as follows: (1) we present a new paradigm for flexibly using proof assistants in developing verified programs; (2) we demonstrate this paradigm using EventML, a new implemented programming/specification language we have built to implement verified distributed protocols; and (3) we report on our experience in using that tool as a *workable new balance* between programming, specifying, and verifying.

2 Illustration of our paradigm: leader election in a ring

Using EventML we have *successfully coded several correct-by-construction distributed protocols* including a two-thirds majority consensus. To illustrate our paradigm, let us consider a simple example: leader election in a ring. We have specified this protocol in EventML, imported it into Nuprl to prove safety and liveness of the protocol, and synthesized correct-by-construction code. (EventML's syntax and semantics are presented in Appendix A.)

The goal of this protocol is to elect a leader among some of the nodes (or locations) of a network, assuming that these nodes can be arranged in a ring [31]. We consider the following setting: asynchronous; unidirectional, i.e., each node can communicate with at most one node; and non-anonymous, i.e., each node has a unique identifier. We assume that these identifiers are totally ordered. Rings can be reconfigured, and the interval during which a ring persists without reconfiguration is called an *epoch*. Not all nodes need to be participating in each epoch.

[1] EventML originates from the E# language, defined in Nuprl, for specifying components of distributed systems [10]. EventML can be downloaded from the following address: `http://www.nuprl.org/software/`.

Informally, the protocol works as follows: At a given epoch, one sets up a ring by assigning a neighbor to each member of the ring[2]. At any point in time, each node can only be participating in a single epoch and have a single neighbor at that epoch. One triggers the election of a leader of a given epoch by sending a request to one of the nodes participating in that epoch. The nodes of the ring vote for a leader by sending proposals to their neighbors. Nodes ignore requests and proposals for epochs different from the epoch they are participating in. A node prompted to elect a leader votes for itself. A node reacts on a proposal by proposing to its direct neighbor the greatest of the following two values: its own identifier and the proposal it has received. The only exception is when a node receives a proposal that proposes its own identifier. In that case, the node declares itself as the leader.

The safety property of this protocol states that there can only be one leader elected at each epoch. Its liveness property states that, at a given epoch, if a node of the ring receives an election request then a leader will eventually be elected (assuming, among other things, that nodes cannot fail, and message delivery is reliable).

Sec. 2.1 presents our EventML specification of this protocol. It illustrates the typical sections that compose an EventML specification, and how programmers can make use of both programming (e.g., operations on data types such as lists or integers) and logical (e.g., verified specifications) concepts from Nuprl's library via import declarations. Because some specifications are only correct in specific contexts, Sec. 2.2 presents how one can make assumptions in EventML to define such contexts. In addition, one can also make assertions. Assumptions and assertions serve two purposes: they provide guidelines for the verification task, and also document the code. Sec. 2.3 presents what one obtains when docking to Nuprl. For example, by docking to Nuprl one obtains correct-by-construction code, but also various abstractions necessary to reason about the specification. Finally, Sec. 2.4 discusses safety and liveness.

2.1 Specification

Let us first give a name to our specification as follows:

```
specification leader_ring
```

Type abstractions. We define epochs and node identifiers as integers. For readability issues, we define the two following type abstractions:

```
type Epoch = Int        type Ident = Int
```

Parameters. EventML specifications can be parametrized. Typically, one defines a protocol without committing to a specific collection of locations. EventML does not provide a location constructor, instead locations are always introduced by parameters. The parameters of leader_ring are as follows (these parameters have to be instantiated in order to run the synthesized code):

```
parameter nodes : Loc Bag       parameter client : Loc      parameter uid : Loc -> Ident
```

The first parameter nodes is the collection of nodes that are part of the network (to prove safety and liveness, we will enforce below that nodes has no repeats). The second parameter client is the location that gets notified when a leader has been elected. The third parameter uid is an ordering function on the locations; nodes are identified by integers.

EventML's library. To prove properties of our specification, we need uid to be injective but we currently have no way of specifying such a restriction directly in EventML. However, we can make the following declaration, and assume uid_inject when proving properties of leader_ring:

[2]The members of the network could be responsible for configuring the rings, but configuration, reconfiguration, and fault-tolerance are outside of the scope of this paper.

```
import inject
let uid_inject = inject ::Loc ::Ident uid;;
```

As mentioned above, EventML can make use of logical/programming concepts defined in Nuprl. The inject function is part of Nuprl's library. It takes three arguments: a type A, a type B, and a function f from A to B; and states that f is an injective function from A to B. We use double colons to write types in expressions.

Nuprl provides a way to export its library in an ASCII file. EventML can then transform this file into an EventML library file, which contains a declaration for every Nuprl object whose type is an EventML type. The following declaration is part of the EventML library file:

```
constant inject (A : Type; B : Type; f : A -> B) : Prop
```

Interface. In EventML, one specifies a distributed system by specifying how its agents interact with each other. Abstractly, agents react to events. An event is a point in space/time with which some information is associated. On some events, agents might try to cause other events to happen at specific locations. In the model of distributed processes underlying EventML [11], an event corresponds to the receipt of a message, the information associated with an event is the data of the corresponding message, and an agent tries to cause an event at location *loc* by sending a message to *loc* (we say "tries" because one cannot enforce message delivery but can only assume it). But in EventML, one does not need to deal with messages, one can simply reason at the more abstract level of events.

The following four kinds of events are involved in our protocol:

```
input config : Epoch * Loc              internal propose : Epoch * Ident
input choose : Epoch                    output   leader  : Epoch * Loc
```

An input config event corresponds to a node being notified of its neighbor at a given epoch. As specified by the type Epoch ∗ Loc, the information associated with such an event is a pair epoch/location. Such an event is called an *input* event because leader_ring's agents can react to such events but cannot cause them to happen. An input choose event corresponds to a node being prompted to start the election of a leader at a given epoch. An internal propose event corresponds to a node receiving a proposal for a node to be elected as the leader of a given epoch. Such an event is called an *internal* event because leader_ring's agents can both react to propose events and try to cause them to happen (i.e., a node can both send and receive proposals). Finally, an output leader event corresponds to a node being notified that a leader has been elected at a given epoch. Such an event is called an *output* event because leader_ring's agents cannot react to such events, but can only try to cause them to happen.

This concludes the definition of the *header* of our specification. In order to define its *body*, we first need to discuss some logical features of EventML.

Interlude—the Logic of Events and event classes. Formally, distributed systems are expressed in a logic called the Logic of Events [10, 12], which is a logical framework implemented in Nuprl to reason about and synthesize distributed protocols. The Logic of Events is outside the scope of this paper, but let us present the concept of an *event class* which is used in the rest of this paper. In the Logic of Events, distributed systems are defined as *event classes*, which are also called *event observers*. Event classes specify information flow in a network of reactive agents by observing the information computed by the agents when events occur—i.e., on receipt of messages. Event classes have two facets: computational and logical. They are functions that associate information (bags of values) with events, and from a computational point of view, they can also be seen as processes reacting to events and producing values. This dualism pervades the logic. As a matter of fact, the event classes discussed in this paper are all implementable by distributed processes, which allows EventML to synthesize processes from specifications. Given this dualism, we allow ourselves to present event classes as processes. We define several event classes below, such as ChooseReply that (sometimes) produces proposals when choose events occur. In the Logic of Events, event classes are built from *event class combinators*, and EventML features logical constructs corresponding to these combinators (such as || or @—see below).

State machines. Let us now define leader_ring's body. As mentioned above, each node keeps track of its neighbor at the epoch it is participating in. To achieve that, each node runs a state machine (a Moore machine) whose transition function is defined as follows:

```
let update loc (epoch, nbr) (epoch', nbr') =
  if epoch > epoch' then (epoch, nbr) else (epoch', nbr');;
```

The first argument of update corresponds to the location at which the state machine is running. Its second argument is a new configuration, and its third argument is the current state of the machine. A node only updates its state when it receives a new configuration for a higher epoch. We define the state machine itself as follows:

```
import State1
class Nbr = State1 (\loc.(0,loc)) update config'input;;
```

The keyword class indicates that Nbr is an event class. We could have used let but it would have been less informative. State1 is a parametrized event class defined in Nuprl, for which several properties have already been proven, and that behaves like a collection of state machines (Moore machines) acting at different locations. State1's parameters are: a function specifying the initial state of the state machine at each location; a transition function; an event class that recognizes inputs triggering state machine transitions. Nbr is defined as an instance of State1: initial states are provided by the function (\loc.(0, loc)), i.e., if Nbr is running at location loc, its initial state is the pair (0,loc), where 0 is considered as a dummy epoch; Nbr's transition function is update; Nbr's inputs are provided by config'input which is an event class implicitly declared along with the config interface. This class observes the information associated with config events whenever such events occur, i.e., it observes the receipt of pairs epoch/location corresponding to new configurations.

Response to an election request. We now specify a small process in charge of responding to election requests, i.e., to choose events. The event class choose'input observes such events, i.e., it returns the epoch at which an election is requested. On receipt of an election request, the process checks that the received epoch is the same as the epoch it is currently participating in, as given by Nbr. If it is, the process proposes itself as the leader to its neighbor, otherwise it does not do anything. The following class specifies that process:

```
class ChooseReply =
  let F loc (epoch, nbr) epoch' =
    if epoch = epoch'
    then {propose'output nbr (epoch, uid loc)}
    else {}
  in F o (Nbr,choose'input) ;;
```

The local function F does exactly what is explained above. The parameter loc corresponds to the location at which the process is running. The parameter (epoch,nbr) is produced by Nbr and epoch' by choose'input. Curly braces are bag delimiters. F's then branch produces only one value which is a proposal addressed to nbr. The identifier propose'output is a function implicitly declared along with the propose interface. The meaning of (propose'output nbr v) is the following instruction: "try to cause a propose event with information v to happen at location nbr". From a message perspective, one can read (propose'output nbr v) as "send a propose message with data v to nbr". F's else branch does not produce any value, meaning that if epoch is not equal to epoch' then ChooseReply does not do anything. The event class F o (Nbr,choose'input) is an instance of what we call the *simple composition combinator*: if that process runs at location loc, the event class F o ($X_1, \ldots, X_n$) produces F loc v_1 ... v_n iff for all $i \in \{1, \ldots, n\}$, X_i produces v_i.

Response to the receipt of a proposal. Next, we specify a small process in charge of responding to proposals, i.e., to propose events. The event class propose'input observes such events, i.e., it returns pairs epoch/identifier of the form (epoch,id) where id corresponds to a node that has been proposed to be the leader of epoch epoch. As in ChooseReply, on receipt of a proposal, the process checks that the received epoch is the same as the epoch it is currently participating in (given by Nbr). If it is, either the process elects itself as the leader if the received identifier is the same as its own id, or sends a new proposal to its neighbor. The following class specifies that process:

```
import imax
class ProposeReply =
  let F loc (epoch, nbr) (epoch', ldr) =
    if epoch = epoch'
    then if ldr = uid loc
         then {leader'output client (epoch, loc)}
         else {propose'output nbr (epoch, imax ldr (uid loc))}
    else {}
  in F o (Nbr,propose'input) ;;
```

Once again we import an abstraction from Nuprl's library: the function imax which computes the maximum of two integers. A ProposeReply process uses imax to compute the maximum between its own identifier and the identifier it has received, and sends that value to its neighbor (in the case where it does not elect itself as the leader). As for ChooseReply, ProposeReply is an instance of the simple composition combinator.

Main program. Finally, each process of the leader election protocol runs both ChooseReply and ProposeReply in parallel and is installed at one of the nodes in nodes:

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

2.2 Hardening leader_ring with assumptions and assertions

Assumptions. To prove leader_ring's safety and liveness properties, in addition to the fact that uid has to be injective, we need to make a few other assumptions. Some of them can be stated in EventML. There are several advantages in stating these assumptions in EventML rather than in Nuprl. As mentioned above, one is that they document the code. In addition to being useful to the programmer, that kind of documentation is also useful to the person in charge of verifying the correctness of the specification, which might not be the same person as the one writing the specification. Another advantage is that EventML provides some automation and its interface is designed to be friendlier.

We assume that at each epoch, a node is not configured twice with different neighbors:

```
let consistent_confs = forall e1 e2: Event. forall nbr1 nbr2: Loc. forall epoch: Epoch.
        config'input observes (epoch,nbr1) @ e1
    => config'input observes (epoch,nbr2) @ e2
    => location e1 = location e2
    => nbr1 = nbr2 ;;
```

The proposition (config'input observes (epoch,nbr1) @ e1) can be read as follows: the class config'input observes/produces (epoch,nbr1) at event e1, i.e., e1 is a config event (occurring at location (location e1)).

We impose event more restrictions on configurations. Let us assume that a location loc can only receive a configuration (epoch,nbr) where epoch is strictly positive; nbr is an agent of the system, i.e., a member of nodes; and loc is not its own neighbor, i.e., loc $\neq$ nbr:

```
import bag–member
let valid_conf = forall e : Event. forall nbr : Loc. forall epoch : Epoch.
       config'input observes (epoch,nbr) @ e
    => epoch > 0  /\  bag–member ::Loc nbr nodes  /\  !(nbr = location e) ;;
```

Finally, let us define ring_setup which specifies what we mean by "At a given epoch, one sets up a ring by assigning a neighbor to each member of the ring":

```
import ring l_member sub–bag no_repeats mk_bag
let ring_setup = forall epoch: Epoch.
    exists L: Loc List. exists succ: Loc -> Loc.
         (forall i : Loc. l_member i L ::Loc => l_member (succ i) L ::Loc)
      /\ sub–bag ::Loc (mk_bag L) nodes
      /\ no_repeats ::Loc L
      /\ ring L succ
      /\ (forall i : Loc. l_member i L ::Loc =>
            exists e : Event.
                   config'input observes (epoch,succ i) @ e
                /\ location e = i
                /\ forall e' : Event. choose'input observes epoch @ e' => e before e')
      /\ (forall i : Loc. forall e : Event.
            config'input observes (epoch,i) @ e => l_member (location e) L ::Loc;;
```

A ring is represented by a location list L and a successor function succ. We assume that L has no repeats and that each element of L belongs to nodes. The function ring cannot currently be expressed in EventML—ring is defined in Nuprl using set/refinement types which are not implemented in EventML. Informally, it states that succ is an injective function from L to L, and that any element of L is accessible from any element of L via succ. We also assume that each node in L eventually gets notified of its participation in the ring, and that no leader election request can be received before every node participating in the ring gets notified. Finally, we enforce that there can only be a single ring set up at each epoch. Note that some restrictions (such as the last one) are not necessary, and only here to simplify the verification task.

State machine properties. In addition to assumptions, one can assert propositions such as state machine properties. An obvious property of Nbr is that it can only produce positive epochs. This property is called an *invariant* and can be stated in EventML as follows:

```
invariant pos_epochs on (epoch,nbr) in Nbr == epoch >= 0 ;;
```

Given such a state machine property, EventML automatically generates the corresponding Logic of Events formula. Once loaded in Nuprl, we have built tactics that try to automatically prove such formulas. Out tactic specialized for proving invariants proves pos_epochs automatically.

2.3 Docking to Nuprl

We can now dock to Nuprl and load leader_ring. The EventML tool set contains a translator that generates for each of our EventML declarations a Nuprl term that expresses its semantic meaning. For example, let us consider what is generated on the Nuprl side for our main declaration.

Abstractions. First of all, our translator generates the following parametrized Nuprl abstraction:

```
leader_ring_main(client;nodes;uid) ==
   leader_ring_ProposeReply(client;uid) || leader_ring_ChooseReply(uid)@nodes
```

Note that EventML computes the necessary parameters: ChooseReply depends only on uid, while ProposeReply depends on uid and client. Because, in addition, our main specification mentions nodes, it depends on nodes, uid, and client.

Well-formedness lemmas. Then, our translator generates the following well-formedness (or typing) lemma:

```
∀[client:Id]. ∀[nodes:bag(Id)]. ∀[uid:Id → ℤ → ℤ].
  (leader_ring_main(client;nodes;uid) ∈ EClass'(Id × Message))
```

From Nuprl's point of view, our main specification produces directed messages which are represented by pairs location/message of the form (loc,msg), representing instructions of the form "send message msg to loc" (Id and Loc are the same type).

Programmability lemmas. Another important lemma that our translator generates is the following lemma called `leader_ring_main_programmable` which states that our main specification is implementable (or programmable):

```
∀client:Id. ∀nodes:bag(Id). ∀uid:Id → ℤ → ℤ.
  Programmable'(Id × Message;leader_ring_main(client;nodes;uid))
```

Extraction. The Nuprl term `TERMOF{leader_ring_main_programmable:o, 1:l, 1:l}` allows one to access the extract of the above programmability lemma. One can think of that term as a pointer to `leader_ring_main_programmable`'s extract. It evaluates to a collection of distributed processes that implement leader_ring. We can then wrap that extract in a Nuprl abstraction as follows:

```
leader_ring_extract(client;nodes;uid) ==
  TERMOF{leader_ring_main_programmable:o, 1:l, 1:l} client nodes uid
```

This abstraction can then be exported and used in EventML. Its well-formedness lemma is trivially the following one:

```
∀client:Id. ∀nodes:bag(Id). ∀uid:Id → ℤ.
  (leader_ring_extract(client;nodes;uid)
  ∈ Programmable'(Id × Message;leader_ring_main(client;nodes;uid)))
```

By unfolding the definition of "programmability", one can prove this other well-formedness lemma, which makes explicit that `leader_ring_extract` is a collection of distributed processes reacting to messages and producing directed messages (where `dataflow` is the type of processes):

```
∀client:Id. ∀nodes:bag(Id). ∀uid:Id → ℤ.
  (leader_ring_extract(client;nodes;uid) ∈ bag(Id × dataflow(Message;bag(Id × Message))))
```

Standard assumptions. In addition to such abstractions and lemmas, EventML generates various abstractions necessary to prove the correctness of a specification.

For example, EventML generates an instance of the following assumption: the output and internal events of a specification S can only be caused by S, which can, e.g., be enforced using a message encryption mechanism. Let leader_ring_message-constraint be the instance of that assumption generated for leader_ring. It assumes that output leader and internal propose events can only be caused by leader_ring. Such assumptions are necessary to prove safety properties.

Similarly, to prove liveness or non-blocking properties we assume that message delivery is reliable, meaning that if a distributed system produces a directed message of the form (loc,msg) then loc will eventually receive msg, causing an event to happen at location loc. Let leader_ring_messages-delivered be the instance of that assumption generated for leader_ring.

2.4 Cooperating with Nuprl to verify leader_ring

Safety. leader_ring's safety property can be stated in EventML as follows:

```
let safety = forall e1 e2 : Event. forall leader1 leader2 : Loc. forall epoch : Epoch.
      leader'input observes leader1 @ e1 /\ leader'input observes leader2 @ e2
   => leader1 = leader2
```

Assuming uid_inject (defined in Sec. 2.1), consistent_confs (defined in Sec. 2.2), valid_conf (defined in Sec. 2.2), ring_setup (defined in Sec. 2.2), and leader_ring_message-constraint (discussed in Sec. 2.3), we have proved in Nuprl that safety is true.

This property cannot be proved automatically by Nuprl, and requires several intermediate lemmas. One can then either harden the EventML specification with extra assertions, or directly work in Nuprl. As mentioned in Sec. 2.2, there are several advantages in stating them in EventML. (Note that currently, neither Nuprl nor EventML provide hints as to what to assert.) Once new assertions have been added to the EventML specification, it can be re-translated into Nuprl to replace the old one, which will not affect the already proved lemmas as long as the code itself does not change. Regarding leader_ring, many such intermediate results were easy to prove thanks to the fact that we have used in our specification several abstractions about which a large number of lemmas have already been proved in Nuprl.

Liveness. To prove leader_ring's liveness property, we made a slightly stronger assumption than ring_setup: we also assumed that once an election has started, nodes will not be reconfigured until the election is finished. Let ring_setup' be the function that captures that property. Using ring_setup', leader_ring's liveness property can be stated in EventML as follows:

```
let liveness = forall e: Event. forall epoch: Epoch. forall L: Loc List.
      length L > 0                        =>   ring_setup' epoch L
   => choose'input observes epoch @ e   =>   l_member (location e) L ::Loc
   => exists e' : Event. exists i : Loc.
        uid i = imax-list(map uid L) /\ leader'input observes (epoch,i) @ e';;
```

We have proved that liveness is true under similar hypotheses as the ones used to prove safety. As mentioned above, an extra assumption we had to make is that messages are reliably delivered.

3 Related work

Tempo. Tempo's approach [32, 35] is related to ours in the sense that it is a framework providing support to specify and run distributed systems, but also to interact with formal method tools, such as model checkers and theorem provers (such as PVS [37]) in order to reason about such systems. Tempo is based on TIOA [25] which itself is based on the IOA [21, 23, 22] programming/specification language for describing asynchronous distributed systems as I/O automata. Although related to our approach, Tempo's approach differs from ours in the sense that its language is non-standard in the programming language community, and to the best of our knowledge it does not support "two-way" collaborative development using theorem provers.

TLA. The TLA temporal logic is based on first-order logic and set theory, and "provides a mathematical foundation for describing systems" [27]. TLA$^+$ [27, 16, 28] is a language for specifying systems described in TLA. TLAPS "is a platform for the development and mechanical verification of TLA$^+$ proofs" [16]. To validate proofs, TLAPS interacts with theorem provers (such as Zenon [14]), proof assistants (such as Isabelle [38]), SMT solvers, and decision procedures. However, TLA$^+$ does not perform program synthesis.

F*. F* [44] is a recent dependently-typed programming language that aims, among other things, at verifying protocols using types expressive enough to formalize and reason about security properties. Part of this expressivity comes from the use of dependent types and (ghost) *refinement types* (types refined by formulas). Ghost refinement types allow one to specify, among other things, cryptographic mechanisms that can be used to ensure security properties. F* features a type checker that collaborates with a SMT solver (Z3) to check the validity of logical formulas and generate proof terms. It is however unclear what level of involvement is required from programmers to interact with Z3 in order to ease type checking.

4 Conclusion and future work

This paper reports on a new "prover assisted" programming paradigm. It illustrates this paradigm using the EventML programing interface we have built and applied to the particular task of writing correct distributed protocols. EventML interfaces with Nuprl to allow programmers to verify their protocols.

Nuprl is especially well-suited to develop distributed systems thanks to its implementation of the Logic of Events, but the ideas expressed in this paper are not restricted to distributed systems and Nuprl. If, say Coq, had an implementation of the Logic of Events, one could extend EventML to interface with Coq. Moreover, as long as they provide enough support and automation, other proof assistants might be more suitable for other kinds of programming. Ideally, EventML programmers would be able to choose among a variety of proof assistants and pick the one they know best or the one for which they can get some help from an expert.

We also plan on connecting EventML to other formal method tools such as SMT solvers and model checkers. More generally, any tool that could potentially help programmers gain some confidence in their code before tackling the task of verification is a good candidate.

We are also considering lightweight formal method tools such as more expressive type systems. For example, extending EventML with some forms of dependent types would move some of the verification from the proof assistant side to the programming language side. However, we want to keep a working balance between (semi-)automatic type inference and a small amount of explicit types necessary to perform type inference.

Finally, proof assistants are becoming more and more widely used in teaching formal methods, and we believe that tools such as EventML would provide a favorable entry point to using proof assistants.

References

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

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

[3] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. `www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`, 2005.

[4] David Aspinall, Serge Autexier, Christoph Lüth, and Marc Wagner. Towards merging platomega and pgip. *Electr. Notes Theor. Comput. Sci.*, 226:3–21, 2009.

[5] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[6] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In *Proceedings of the 2004 Int'l Conf. on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] Stefan Berghofer. Program extraction in simply-typed higher order logic. In *Types for Proofs and Programs, Second Int'l Workshop, TYPES 2002*, volume 2646 of *LNCS*, pages 21–38. Springer, 2002.

[8] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Types for Proofs and Programs, Int'l Workshop, TYPES 2000*, volume 2277 of *LNCS*, pages 24–40. Springer, 2000.

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

[10] 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.

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

[12] 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.

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

[14] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th Int'l Conf., LPAR 2007*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[15] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd Int'l Conf.*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

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

[17] Adam Chlipala. A verified compiler for an impure functional language. In *37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–106. ACM, 2010.

[18] 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.

[19] Robert L. Constable, Scott Johnson, and C. D. Eichenlaub. *An Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science*. Springer, 1982.

[20] Catarina Coquand and Thierry Coquand. Structured type theory. In *In Proc. Workshop on Logical Frameworks and Meta-languages*, 1999.

[21] S. Garland and N. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1998.

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

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

[24] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP*, volume 2618 of *LNCS*, pages 284–301. Springer, 2003.

[25] Dilsun Kirli Kaynar, Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. *The Theory of Timed I/O Automata, Second Edition*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

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

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

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

[29] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 42–54. ACM, 2006.

[30] 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.

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

[32] Nancy A. Lynch, L. Michel, and A. A. Shvartsman. Tempo: A toolkit for the timed input/output automata formalism. In *Proceedings of the 1st Int'l Conf. on Simulation Tooks and Techniques for Communications, Networks and Systems – Industrial Track: Simulation Works*, 2008.

[33] Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming, 5th Int'l School*, volume 3622 of *LNCS*, pages 130–170. Springer, 2004.

[34] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[35] Peter Musial. Using timed input/output automata for implementing distributed systems. Technical report, Cambridge, 2011.

[36] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[37] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. Pvs: Combining specification, proof checking, and model checking. In *Computer Aided Verification, 8th Int'l Conf.*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.

[38] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[39] Lee Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Formal Methods in Computer-Aided Design, 7th Int'l Conf.*, pages 231–238. IEEE Computer Society, 2007.

[40] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[41] Vincent Rahli. *Investigations in intersection types: Confluence, and semantics of expansion in the λ-calculus, and a type error slicing method*. PhD thesis, Heriot-Watt University, January 2011.

[42] John M. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Trans. Software Eng.*, 25(5):651–660, 1999.

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

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

[45] Marc Wagner, Serge Autexier, and Christoph Benzmüller. Platomega: A mediator between text-editors and proof assistance systems. *Electr. Notes Theor. Comput. Sci.*, 174(2):87–107, 2007.

Figure 1 core EventML syntax

(a) functional part

n	$\in$ Nat		(natural numbers)
vid	$\in$ Vid		(value identifiers)
a	$\in$ TyVar		(type variables)
ptc	$\in$ PTyC	$::=$	`Int` $\mid$ `List` $\mid$ `Bool` $\mid$ `Unit`
itc	$\in$ ITyC	$::=$	$*$ $\mid$ $\rightarrow$ $\mid$ $+$
c	$\in$ Const	$=$	`true` $\mid$ `false` $\mid$ `nil`
op	$\in$ Op	$::=$	`+` $\mid$ `-` $\mid$ `=` $\mid$ `.` $\mid$ `++` $\mid$ `<` $\mid$ `>` $\mid$ `or` $\mid$ `&`
exp	$\in$ Exp	$::=$	$vid \mid n \mid c \mid {\sim}exp \mid exp\,\texttt{:}\,ty$
			$\mid (exp_1, \ldots, exp_n)$
			$\mid exp_1\ op\ exp_2$
			$\mid exp_1\ exp_2$
			$\mid \backslash pat\,.\,exp$
			$\mid$ `if` exp_1 `then` exp_2 `else` exp_3
			$\mid$ `let` $bind$ `in` exp
pat	$\in$ Pat	$::=$	$vid \mid \_ \mid (pat_1, \ldots, pat_n) \mid pat\,\texttt{:}\,ty$
$tyseq$	$\in$ TySeq	$::=$	$\epsilon \mid ty \mid (ty_0, \ldots, ty_n)$
ty	$\in$ Ty	$::=$	$a \mid tyseq\ ptc \mid ty_1\ itc\ ty_2 \mid (ty)$
$bind$	$\in$ Bind	$::=$	$vid\ pat_1\ \cdots\ pat_n$ `=` exp
dec	$\in$ Dec	$::=$	`let` $bind$`;;`
$prog$	$\in$ Prog	$::=$	$dec\ \ulcorner prog \urcorner$

(b) logic part

PTyC	$::=$	$\cdots$ $\mid$ `Bag` $\mid$ `Class` $\mid$ `Loc`
		$\mid$ `Prop` $\mid$ `Event` $\mid$ `Inst`
Op	$::=$	$\cdots$ $\mid$ `>>=` $\mid$ `||` $\mid$ `@` $\mid$ `before` $\mid$ `/\` $\mid$ `\/`
Exp	$::=$	$\cdots \mid \{exp_1; \ldots; exp_n\}$
		$\mid$ `Prior(`exp`)`
		$\mid exp$ `o` $(exp_1, \cdots, exp_n \ulcorner$`,Prior(self)?`$exp'\urcorner)$
		$\mid$ `forall` vid `:` ty`.` exp
		$\mid$ `exists` vid `:` ty`.` exp
		$\mid exp_1$ `observes` exp_2 `@` exp_3
Dec	$::=$	$\cdots$ $\mid$ `specification` vid
		$\mid$ `parameter` vid `:` ty
		$\mid$ `import` $vid_0\ \cdots\ vid_n$
		$\mid$ `internal` vid `:` ty
		$\mid$ `input` vid `:` ty
		$\mid$ `output`vid `:` ty
		$\mid$ `main` exp

A EventML's syntax and static semantics

A.1 Syntax

Core EventML. EventML's syntax is based on Classic ML's syntax and the Logic of Events [10, 12]. This section presents a core subset of EventML's syntax called *core EventML*. We divide this presentation into two parts: Fig. 1a presents its ML part and Fig 1b presents its Logic of Events part. In these two figures we write $\ulcorner x \urcorner$ to indicate that x is optional. These brackets are not part of EventML's syntax. For example, a program *prog* can either be a declaration, or a declaration followed by another program (it allows us to define recursive production rules).

Core EventML's functional part. First, let us describe the functional part of core EventML. The set Op is a binary infix operator set: `+` and `-` are the usual addition and subtraction operators on integers; `.` is the list constructor, `++` is the list appending; `<` and `>` are the usual integer comparison operators; `or` and `&` are the Boolean operators "or" and "and". We use round parentheses to build tuples. The expression `let` *bind* `in` *exp* allows one to locally bind identifiers (e.g., local functions). The expression $\backslash pat\,.\,exp$ is a lambda-expression. The types of expressions can be explicitly constrained as follows: $exp\,\texttt{:}\,ty$. Similarly, patterns' types can be constrained as follows: $pat\,\texttt{:}\,ty$. The pattern $\_$ is the wildcard pattern.

An EventML type can either be: a type variable a, a type construction *tyseq ptc* (e.g., the integer list type `Int List`), a disjoint union type $ty_1\ +\ ty_2$, a function type $ty_1 \rightarrow ty_2$, a product type $ty_1\ *\ ty_2$, or a parenthesized type (ty). The symbol ϵ is our internal notation for the empty type sequence. For example, formally, the integer type is the type construction ($tyseq$ `Int`) where *tyseq* is the empty type sequence.

We impose the following syntactic restriction on binders: no identifier can occur twice in the $atpat_i$'s of a binder of the form $vid\ atpat_1\ \cdots\ atpat_n$ `=` exp. Under this condition, a binder of the form $vid\ pat_0\ \cdots\ pat_n$ `=` exp has the same meaning as vid `=` $\backslash pat_0 . \ldots . \backslash pat_n . exp$. The first form allows one to define functions without lambda expressions.

Core EventML's logical part. Fig. 1b adds a few binary operators to the language. The operator `>>=` corresponds to the delegation primitive combinator of the Logic of Events. This combinators allows processes to delegate tasks to sub-processes. The operator `||` corresponds to the non-primitive parallel combinator of the Logic of Events. This combinator allows one to run two processes in parallel. The operator `@` corresponds to the non-primitive "at" combinator of the Logic of Events. This combinator restricts the locations at which processes are running. The operator `before` is an operator on events: exp_1 `before` exp_2 is true iff exp_1 is an event that occurs causally before the event exp_2. The two operators `/\` and `\/` are the usual "and" and "or" operators on propositions. Fig. 1b also adds three forms of expressions. We use curly braces for bags, e.g., $\{1;2\}$ is a bag and is equal to the bag $\{2;1\}$ but not to the bag $\{1;2;2\}$. The `Prior`(exp) form corresponds to the "prior" primitive combinator of the Logic of Events. This combinator allows one to observe past events. In terms of computation, `Prior`(exp) produces values produced by exp in the past. The third production rule effectively allows one to write forms corresponding to both the simple (or non-recursive) and recursive composition combinators of the Logic of Events. The simple composition combinator allows one to combine the observations of a list of classes to form a new observation. For example, leader_ring specified in Sec. 2.1 makes use of the simple composition combinator to define ChooseReply and ProposeReply. Given an observation (epoch,nbr) made by Nbr at location loc and an observation epoch' made by choose'input at the same location, the event class ChooseReply observes either (propose'output nbr (epoch, uid loc)) if epoch = epoch' and nothing otherwise. The expression (propose'output nbr (epoch, uid loc)) has type `Inst`—which stands for "Instruction"—and can be seen as the instruction: "try to cause a propose event with value (epoch, uid loc) at location nbr", or the instruction "try send the proposition (epoch, uid loc) to nbr". The recursive composition combinators is similar to the simple one but in addition allows one to make use of prior observations of the combinator. For example, the event class (State1 init upd cls) is defined as follows: (upd o (cls, Prior(self)?(\loc.{ init loc}))). This class specifies a collection of state machines (Moore machines) acting at different locations whose inputs are given by cls, whose initial state are given by init, and whose transition function is upd. Expressions of the form `forall` *vid* `:` *ty*`.` *exp* and `exists` *vid* `:` *ty*`.` *exp* are the usual universally and existentially quantified statements. Finally, exp_1 `observes` exp_2 `@` exp_3 is a proposition expressing that the class exp_1 observes exp_2 at event exp_3.

One can name a specification using a declaration of the form `specification` *vid*.

Protocols can be parametrized. For example, typically, one defines a protocol without committing to a specific location set. EventML does not provide a location constructor, instead locations are always given by parameters such as: `parameter` *locs* `: Loc Bag`.

In EventML one can import Nuprl abstractions, stored in a library file (a snapshot of Nuprl's library of abstractions, lemmas, and extracts), using `import` declarations. Many operations on lists (mapping, filtering, etc.), bags, integers, etc., are available, but also event classes such as state machines (necessary to define processes requiring memory), etc.

When specifying a protocol, one has to specify its interface, i.e., the kinds of messages its agents act upon or send. More abstractly, it corresponds to the kinds of events that can happen in the system. A message/event kind declaration consists of a status, an identifier and a type. An event can have one of these three statuses: `internal`, `input`, or `output`. The status differs as follows: a protocol should not cause input events and should not react to output events. For example, Sec. 2.1 declares the following interface: input choose : Epoch, which means that the agents of the specified system can cause, but not react to, choose events. Some information is associated with each event. For example, with each choose event is associated an epoch (from a more practical point of view, one can think of that information as the data of a message).

To observe the information associated with each input and internal event, EventML declares an "input" class, also called a *base* class, along with each `input` and `internal` declaration. Such base classes allow processes to react to the corresponding events. An "output" function is implicitly defined for each `internal` and `output` event declaration, in order to trigger such events. The names of such a bases classes ans functions are based on the identifier part of the corresponding event declaration.

Finally, a specification is an expression that can be declared as such as follows: `main` *exp*. EventML generates programmability lemmas for such `main` declarations, from which Nuprl can synthesize code. A program extracted from the proof of such a programmability lemma satisfies (is an implementation of) the corresponding specification by construction.

A.2 Static semantics

This section presents EventML's static semantics but does not define EventML's dynamic semantics. One cannot evaluate an EventML specification. One has to first synthesize processes from a specification, and then to run these processes, one has to use any of our evaluators.

Note that our implemented type inferencer is constraint based, but this section presents EventML static semantics using a more "traditional" typing relation.

First, let us define the syntax of internal types. Internal types are similar to the external types defined in Fig. 1a. We do not distinguish between internal and external type constructor (postfix and infix). We reuse the type variables introduced in Fig. 1a, but we also introduce a new kind of type variables called equality type variables. The type variables introduced in Fig. 1a 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 on which equality is decidable (e.g., integers, Booleans, etc., but not functions—this is formally defined below). Internal types, type schemes, and type environments are defined as follows (we use angle brackets for tuples):

$$\begin{array}{lll} ea \in \mathsf{EqTyVar} & & \text{(equality type variables)} \\ \alpha \in \mathsf{ITyVar} & ::= & a \mid ea \\ \overrightarrow{\tau} \in \mathsf{ITySeq} & ::= & \langle \tau_1, \ldots, \tau_n \rangle \\ \tau \in \mathsf{ITy} & ::= & \alpha \mid \overrightarrow{\tau}\ ptc \mid \tau_1\ itc\ \tau_2 \end{array} \qquad \begin{array}{lll} \overline{\alpha} \in \mathsf{ITyVarSet} & ::= & \{\alpha_1, \ldots, \alpha_n\} \\ \sigma \in \mathsf{ITyScheme} & ::= & \forall \overline{\alpha}.\ \tau \\ \Gamma \in \mathsf{ITyEnv} & = & \mathsf{Vid} \to \mathsf{ITyScheme} \end{array}$$

Type schemes are subject to α-conversion (equality type variables can only be renamed to equality type variables, and similarly for non-equality type variables). In type environments, we sometimes write: τ for the type scheme $\forall\varnothing.\ \tau$, and ptc for the internal type $\langle\rangle\ ptc$.

Let $\Gamma_1 + \Gamma_2 = \Gamma_2 \cup \{vid \mapsto \Gamma_1(vid) \mid vid \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2)\}$ (this operator is used to type check expressions with binders such as lambda or let expressions). Let $\Gamma_1 \uplus \Gamma_2$ be $\Gamma_1 \cup \Gamma_2$ if $\Gamma_1 \cup \Gamma_2$ is a function and undefined otherwise (this operator is used to type check patterns).

Let us define the "admits equality" predicate on internal type constructors and internal types. We start by defining this predicates on internal type constructors as follows:

$$\begin{array}{lcl} \mathsf{admitsEq}(ptc) & \iff & ptc \notin \{\texttt{Class}, \texttt{Prop}, \texttt{Inst}\} \\ \mathsf{admitsEq}(itc) & \iff & itc \neq \to \end{array}$$

A type constructor admits equality iff equality is decidable in that type. There is no equality decider for functions. Also, because classes are functions (from runs and events to bags of values), there is no equality decider for classes.

Let the "admits equality" predicate be defined on internal types as follows (by simple induction on the structure of its argument):

$$\begin{array}{lll}
\mathsf{admitsEq}(\tau) & \iff & \begin{cases} \quad \tau = ea \\ \lor\ \tau = \overrightarrow{\tau}\ ptc \land \mathsf{admitsEq}(ptc) \land \mathsf{admitsEq}(\overrightarrow{\tau}) \\ \lor\ \tau = \tau_1\ itc\ \tau_2 \land \mathsf{admitsEq}(itc) \land \mathsf{admitsEq}(\langle \tau_1, \tau_2 \rangle) \end{cases} \\
\mathsf{admitsEq}(\langle \tau_1, \ldots, \tau_n \rangle) & \iff & \forall i \in \{1, dots, n\}.\ \mathsf{admitsEq}(\tau_i)
\end{array}$$

We define substitutions as follows (where dom is the usual domain operator on functions):

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

Substitutions are applied to internal types as follows:

$$\begin{array}{ll}
\alpha[sub] & = \begin{cases} sub(\alpha), \text{ if } \alpha \in \mathsf{dom}(sub) \\ \alpha, \qquad \text{otherwise} \end{cases} \\
(\langle \tau_1, \ldots, \tau_n \rangle\ ptc)[sub] & = \langle \tau_1[sub], \ldots, \tau_n[sub] \rangle\ ptc \\
(\tau_1\ itc\ \tau_2)[sub] & = \tau_1[sub]\ itc\ \tau_n[sub]
\end{array}$$

Type schemes are instantiated as follows:

$$\tau \prec \forall \overline{\alpha}.\ \tau' \iff \exists sub.\ (\tau = \tau'[sub] \land \mathsf{dom}(sub) = \overline{\alpha})$$

We use this relation to provide the static semantics of identifier expressions and of binary operators (see Fig. 2 below).

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

$$\mathsf{fv}(\tau) = \{\alpha \mid \alpha \text{ occurs in } \tau\} \quad \text{and} \quad \mathsf{fv}(\Gamma) = \{\alpha \mid \Gamma(vid) = \forall \overline{\alpha}.\ \tau \land \alpha \in \mathsf{fv}(\tau) \setminus \overline{\alpha}\}$$

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 a type environment into a polymorphic one, i.e., whenever possible types are promoted 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 provide the static semantics of let expressions and let declarations (see Fig. 2 and Fig. 5 below).

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

$$\begin{array}{ll}
\{\ \texttt{+} & \mapsto \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
,\ \texttt{-} & \mapsto \texttt{Int} \to \texttt{Int} \to \texttt{Int} \\
,\ \texttt{=} & \mapsto \forall\{ea\}.\ ea \to ea \to \texttt{Bool} \\
,\ \texttt{.} & \mapsto \forall\{a\}.\ a \to a\ \texttt{List} \to a\ \texttt{List} \\
,\ \texttt{++} & \mapsto \forall\{a\}.\ a\ \texttt{List} \to a\ \texttt{List} \to a\ \texttt{List} \\
,\ \texttt{<} & \mapsto \texttt{Int} \to \texttt{Int} \to \texttt{Bool} \\
,\ \texttt{>} & \mapsto \texttt{Int} \to \texttt{Int} \to \texttt{Bool} \\
,\ \texttt{or} & \mapsto \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \\
,\ \texttt{\&} & \mapsto \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \\
,\ \texttt{@} & \mapsto \forall\{a\}.\ a\ \texttt{Class} \to \texttt{Loc Bag} \to a\ \texttt{Class} \\
,\ \texttt{||} & \mapsto \forall\{a\}.\ a\ \texttt{Class} \to a\ \texttt{Class} \to a\ \texttt{Class} \\
,\ \texttt{>>=} & \mapsto \forall\{a, a'\}.\ a\ \texttt{Class} \to (a \to a'\ \texttt{Class}) \to a'\ \texttt{Class} \\
,\ \texttt{before} & \mapsto \texttt{Event} \to \texttt{Event} \to \texttt{Prop} \\
,\ \texttt{/\textbackslash} & \mapsto \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop} \\
,\ \texttt{\textbackslash/} & \mapsto \texttt{Prop} \to \texttt{Prop} \to \texttt{Prop} \\
\}
\end{array}$$

Figure 2 EventML's static semantics—expressions

$$\frac{\tau \prec \mathit{\Gamma}(\mathit{vid})}{\mathit{vid} : \langle \mathit{\Gamma}, \tau \rangle} \quad \frac{b \in \{\mathtt{true}, \mathtt{false}\}}{b : \langle \mathit{\Gamma}, \mathtt{Bool} \rangle} \quad \frac{}{\mathtt{nil} : \langle \mathit{\Gamma}, \tau\ \mathtt{List} \rangle} \quad \frac{}{n : \langle \mathit{\Gamma}, \mathtt{Int} \rangle} \quad \frac{}{() : \langle \mathit{\Gamma}, \mathtt{Unit} \rangle} \quad \frac{\mathit{exp} : \langle \mathit{\Gamma}, \tau \rangle \quad \mathit{ty} :_{\mathrm{t}} \tau}{\mathit{exp} : \mathit{ty} : \langle \mathit{\Gamma}, \tau \rangle}$$

$$\frac{\mathit{exp} : \langle \mathit{\Gamma}, \mathtt{Int} \rangle}{\sim\mathit{exp} : \langle \mathit{\Gamma}, \mathtt{Int} \rangle} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ \mathit{exp}_i : \langle \mathit{\Gamma}, \tau_i \rangle}{(\mathit{exp}_1, \ldots, \mathit{exp}_n) : \langle \mathit{\Gamma}, \tau_1\ *\ \cdots *\ \tau_n \rangle}$$

$$\frac{\mathit{exp}_1 : \langle \mathit{\Gamma}, \tau_1 \to \tau_2 \rangle \quad \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau_1 \rangle}{\mathit{exp}_1\ \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau_2 \rangle} \qquad \frac{\mathit{pat} :_{\mathrm{p}} \langle \mathit{\Gamma}', \tau \rangle \quad \mathit{exp} : \langle \mathit{\Gamma} + \mathit{\Gamma}', \tau' \rangle}{\backslash \mathit{pat} \,.\, \mathit{exp} : \langle \mathit{\Gamma}, \tau \to \tau' \rangle}$$

$$\frac{\mathit{exp}_1 : \langle \mathit{\Gamma}, \mathtt{Bool} \rangle \quad \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau \rangle \quad \mathit{exp}_3 : \langle \mathit{\Gamma}, \tau \rangle}{\mathtt{if}\ \mathit{exp}_1\ \mathtt{then}\ \mathit{exp}_2\ \mathtt{else}\ \mathit{exp}_3 : \langle \mathit{\Gamma}, \tau \rangle} \qquad \frac{\mathit{bind} :_{\mathrm{d}} \langle \mathit{\Gamma}, \mathit{\Gamma}' \rangle \quad \mathit{exp} : \langle \mathit{\Gamma} + \mathsf{clos}_{\mathit{\Gamma}}(\mathit{\Gamma}'), \tau \rangle}{\mathtt{let}\ \mathit{bind}\ \mathtt{in}\ \mathit{exp} : \langle \mathit{\Gamma}, \tau \rangle}$$

$$\frac{\mathit{exp}_1 : \langle \mathit{\Gamma}, \tau_1 \rangle \quad \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau_2 \rangle \quad (\tau_1 \to \tau_2 \to \tau_3) \prec \mathit{\Gamma}(\mathit{op})}{\mathit{exp}_1\ \mathit{op}\ \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau_3 \rangle} \qquad \frac{\forall i \in \{1, \ldots, n\}.\ \mathit{exp}_i : \langle \mathit{\Gamma}, \tau \rangle}{\{\mathit{exp}_1; \ldots; \mathit{exp}_n\} : \langle \mathit{\Gamma}, \tau\ \mathtt{Bag} \rangle}$$

$$\frac{\mathit{exp} : \langle \mathit{\Gamma}, \tau\ \mathtt{Class} \rangle}{\mathtt{Prior}(\mathit{exp}) : \langle \mathit{\Gamma}, \tau\ \mathtt{Class} \rangle} \qquad \frac{\mathit{exp}_1 : \langle \mathit{\Gamma}, \tau\ \mathtt{Class} \rangle \quad \mathit{exp}_2 : \langle \mathit{\Gamma}, \tau \rangle \quad \mathit{exp}_3 : \langle \mathit{\Gamma}, \mathtt{Event} \rangle}{\mathit{exp}_1\ \mathtt{observes}\ \mathit{exp}_2\ \mathtt{@}\ \mathit{exp}_3 : \langle \mathit{\Gamma}, \mathtt{Prop} \rangle}$$

$$\frac{\begin{array}{c}\mathit{exp} : \langle \mathit{\Gamma}, \mathtt{Loc} \to \tau_1 \to \cdots \to \ulcorner \tau_{n+1} \to \urcorner\ \tau_{n+1} \rangle \qquad \ulcorner \mathit{exp}' : \langle \mathit{\Gamma}, \mathtt{Loc} \to \tau_{n+1}\ \mathtt{Bag} \rangle \urcorner \\ \forall i \in \{1, \ldots, n\}.\ \mathit{exp}_i : \langle \mathit{\Gamma}, \tau_i\ \mathtt{Class} \rangle\end{array}}{\mathit{exp}\ \mathtt{o}\ (\mathit{exp}_1, \cdots, \mathit{exp}_n \ulcorner, \mathtt{Prior}(\mathtt{self})?\mathit{exp}' \urcorner) : \langle \mathit{\Gamma}, \tau_{n+1}\ \mathtt{Class} \rangle}$$

$$\frac{\mathit{ty} :_{\mathrm{t}} \tau \quad \mathit{exp} : \langle \mathit{\Gamma} + \{\mathit{vid} \mapsto \tau\}, \mathtt{Prop} \rangle \quad \mathsf{fv}(\tau) = \varnothing}{\mathtt{forall}\ \mathit{vid}\ \mathtt{:}\ \mathit{ty}\ \mathtt{.}\ \mathit{exp} : \langle \mathit{\Gamma}, \mathtt{Prop} \rangle} \qquad \frac{\mathit{ty} :_{\mathrm{t}} \tau \quad \mathit{exp} : \langle \mathit{\Gamma} + \{\mathit{vid} \mapsto \tau\}, \mathtt{Prop} \rangle}{\mathtt{exists}\ \mathit{vid}\ \mathtt{:}\ \mathit{ty}\ \mathtt{.}\ \mathit{exp} : \langle \mathit{\Gamma}, \mathtt{Prop} \rangle}$$

Figure 3 EventML's static semantics—patterns

$$\frac{}{\mathit{vid} :_{\mathrm{p}} \langle \{\mathit{vid} \mapsto \tau\}, \tau \rangle} \qquad \frac{}{\_ :_{\mathrm{p}} \langle \{\}, \tau \rangle} \qquad \frac{\mathit{pat} :_{\mathrm{p}} \langle \mathit{\Gamma}, \tau \rangle \quad \mathit{ty} :_{\mathrm{t}} \tau}{\mathit{pat} : \mathit{ty} :_{\mathrm{p}} \langle \mathit{\Gamma}, \tau \rangle}$$

$$\frac{\forall i \in \{0, \ldots, n\}.\ \mathit{pat}_i :_{\mathrm{p}} \langle \mathit{\Gamma}_i, \tau_i \rangle}{(\mathit{pat}_0, \ldots, \mathit{pat}_n) :_{\mathrm{p}} \langle \mathit{\Gamma}_0 \uplus \cdots \uplus \mathit{\Gamma}_n, \tau_0\ *\ \cdots *\ \tau_n \rangle} \qquad \frac{}{() :_{\mathrm{p}} \langle \varnothing, \mathtt{Unit} \rangle}$$

The only special case in the definition of this initial environment is the `=` case. In the case of the equality binary operator, we enforce that its arguments have to "admit equality" using an equality type variable (ea) instead of a non-equality type variable (a). This prevents from using the equality binary operator on, e.g., functions.

Let $\mathsf{\Gamma lib}$ be an environment that associates a type scheme with each abstraction declared in EventML's library.

Finally, Fig. 2 presents the static semantics of EventML's expressions. It defines the relation ($\mathit{exp} : \langle \mathit{\Gamma}, \tau \rangle$). Fig. 3 presents the static semantics of EventML's patterns. It defines the relation ($\mathit{pat} :_{\mathrm{p}} \langle \mathit{\Gamma}, \tau \rangle$). Fig. 4 presents the static semantics of EventML's external types. It defines the relations ($\mathit{tyseq} :_{\mathrm{s}} \overrightarrow{\tau}$) and ($\mathit{ty} :_{\mathrm{t}} \tau$). Fig. 5 presents the static semantics of EventML's bindings and declarations. It defines the relations ($x :_{\mathrm{h}} \langle \mathit{\Gamma}, \langle \mathit{\Gamma}', \tau \rangle \rangle$) where x can either be a *hdropt* or a *hdropts*, and ($x :_{\mathrm{d}} \langle \mathit{\Gamma}, \mathit{\Gamma}' \rangle$) where x can either be a *hdr*, or a *bind*, or a *dec*, or a *prog*.

A piece of code *prog* is a valid EventML program iff there exists a type environment $\mathit{\Gamma}$

Figure 4 EventML's static semantics—types

$$\frac{}{\epsilon :_{\mathrm{s}} \langle\rangle} \qquad \frac{ty :_{\mathrm{t}} \tau}{ty :_{\mathrm{s}} \langle\tau\rangle} \qquad \frac{\forall i \in \{1,\dots,n\}.\ ty_i :_{\mathrm{t}} \tau_i}{(ty_0,\dots,ty_n) :_{\mathrm{s}} \langle\tau_1,\dots,\tau_n\rangle}$$

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

Figure 5 EventML's static semantics—bindings and declarations

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

$$\frac{bind :_{\mathrm{d}} \langle\Gamma,\Gamma'\rangle}{\texttt{let}\ bind\texttt{;;} :_{\mathrm{d}} \langle\Gamma, \mathsf{clos}_{\Gamma}(\Gamma')\rangle} \qquad \frac{ty :_{\mathrm{t}} \tau \quad \mathsf{fv}(\tau) = \varnothing}{\texttt{parameter}\ vid\ \texttt{:}\ ty :_{\mathrm{d}} \langle\Gamma, \{vid \mapsto \tau\}\rangle}$$

$$\frac{\forall i \in \{0,\dots,n\}.\ \Gamma(vid_i) = \sigma_i}{\texttt{import}\ vid_0\ \cdots\ vid_n :_{\mathrm{d}} \langle\Gamma, \{vid_0 \mapsto \sigma_0\} + \cdots + \{vid_n \mapsto \sigma_n\}\rangle}$$

$$\frac{status \in \{\texttt{internal}, \texttt{input}, \texttt{output}\} \quad ty :_{\mathrm{t}} \tau}{status\ vid\ \texttt{:}\ ty :_{\mathrm{d}} \langle\Gamma, \{vid'\texttt{input} \mapsto \tau\ \texttt{Class}, vid'\texttt{output} \mapsto \texttt{Loc} \to \tau \to \texttt{Inst}\}\rangle}$$

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

such that $prog :_{\mathrm{d}} \langle \mathsf{\Gamma lib} + \mathsf{\Gamma op}, \Gamma\rangle$ can be derived from the rules presented in Fig. 2 to Fig. 5. The environment Γ is *prog*'s static semantics. It corresponds to the well-formedness lemmas generated when interpreting EventML to Nuprl. We impose that $\mathsf{fv}(\Gamma) = \varnothing$.