The Logic of Events, a framework to reason about distributed systems

Mark Bickford  Vincent Rahli  Robert Constable
Cornell University

Protocol specification and verification. In response to the need of ensuring correctness and security properties of distributed systems and system components in general, programming languages have been developed, featuring rich types such as dependent types or session types, expressive enough to specify such properties. Also, some progress has been done towards automatically proving that programs meet such specifications (often by type checking). Following this line of work, we use constructive logic to synthesize protocols from specifications, and provide tools that facilitate the use of formal methods to prove the correctness of distributed protocols based on asynchronous message passing.

Constructive logic. Constructive logic is appropriate for specifying and reasoning about distributed algorithms for two reasons. The first is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-programs correspondence holds for constructive logic and we want to extract correct-by-construction algorithms and protocols from our proofs. The second reason is that the proofs-as-program

The Logic of Events. We present a logical framework to reason about distributed systems called the Logic of Events. This logic has been formalized in Nuprl [1, 2]. Nuprl’s logic is a constructive type theory called Computational Type Theory (CTT). This logic allows one to reason about events and how they relate to each other via, among other things, a well-founded causal ordering on them. The Logic of Events is based on a model of message passing. An event is an abstract object corresponding to the receipt of a message at a location; the message is called primitive information of the event. A message is a triple of the following dependent type: \( \text{Header} \times T \times Type \times T \). For example, \((\" this is a header\", Z, 1)\) is a message. Event orderings provide the basic structure to reason about events. An event ordering consists of a set of events, a function that associates a location with each event, a function that associates a primitive information with each event, and a causal ordering relation on events. An event ordering can be seen as a formalization of the message sequence diagrams used by protocol designers. A fundamental method for reasoning about properties of event orderings, originally pioneered by Lamport [3], is induction on the causal order. With this method we can prove both safety and liveness properties of our specifications.

Event classes. A central concept in the Logic of Events is the notion of event classes, also called event observers. Event classes observe how distributed systems agents “react” on receipt of messages. Formally, an event class of type \( T \), for some type \( T \), is a function that takes an event ordering and an event in that event ordering, and returns a bag (or multiset) of elements of type \( T \). If the class \( X \) associates the bag \( \{ v_1, \ldots, v_n \} \) with the event \( e \), we say that \( X \) observes the values \( v_i \)'s at \( e \). Typically, one can observe complex information by composing simple event classes. For example, a class may observe that the receipt of a certain message means that consensus has been reached. Note that different classes may observe different values at a single event. Event classes can be seen as having two facets: a logical one and a programming one. By that we mean that, (1) each event class \( X \) can be described by expressing the relation between the elements observed by \( X \) and the elements observed by \( X \)'s components (logical aspect); (2) one can extract programs from those classes we call programmable (programming aspect). Informally a class \( X \) is programmable iff there exists a corresponding program that can produce all and only the elements observed by \( X \). Such extracted programs are implementations of the corresponding classes.

Specifying and proving protocols. We developed a suite of tools and tactics in Nuprl to reason about event classes. We also developed a programming language called EventML which allows programmers to write specifications of distributed protocols. Such a specification is an event class describing the information flow of a distributed program. EventML features an automatic program synthesizer. In addition, it can dock to Nuprl (EventML is interpreted to Nuprl) in order to formally prove protocol properties and generate (within Nuprl) correct-by-construction programs by extraction; we say that these extracted programs are correct-by-docking.

We have specified several protocols such as a simple two-thirds consensus protocol and Paxos, and we keep improving our Nuprl tools and tactics as we prove properties of these protocols. A simple example. Here is a toy example that we call the “ping-pong” specification (written in EventML):

```
open param p : Loc
param locs : Loc Bag

internal ping : "ping" Loc
internal pong : "pong" Loc
input start Loc
output out Loc

import bag-map

class ReplyToPong client loc =
  let F = j = if loc=j then \{out\send\client\loc\} else {} in F \circ pong\base::;

class SendPing loc = Output(\slf. (pong\send\loc\slf) \circ pong\base::);

class Handler (client loc) = SendPing loc || ReplyToPong client loc::

class ReplyToPing =
  (\slf. \loc. (pong\send\loc\slf) \circ pong\base::);

class Delegate =
  let F = client = bag-map (\loc. (client loc) \locs in F \circ start\base::);

main (Delegate >>> Handler) \{p\} || ReplyToPing \{locs
```
Similar to IO-Automata [4] and other specification languages we have declared the input, output, and internal events. Unlike IO-
Automata, there are no explicit state variables. Instead we have de-
clared the base classes \( \text{start base, ping base, and pong base} \) that
observe the receipt of input and internal messages (with headers
`start`, `ping`, and `pong` respectively) and we have de-
defined other classes (Delegate, Handler, SendPing, ReplyToPong,
and ReplyToPing) in terms of the base classes using class combinators.

The combinator \( X \parallel Y \) is a parallel composition of classes
\( X \) and \( Y \). An expression like \( \text{cout s loc} \) pong is a function
composition, applying the function \( \text{cout s loc} \) to the information
observed by class Pong.

Our tool provides a language for ele-
mentary specification of distributed algorithms that use asyn-
chronous message passing. It is not yet suited for reasoning about
concurrent shared memory systems.

It provides automated code synthesis that is correct-by-construction.
It also automates major parts of the reasoning about higher-level
and liveness properties of the specified systems, by automatical-
genrating the inductive logical form. We are working on
methods for automatic verification of invariants.

The theory underlying these tools is quite mature and the tools
have already been used to carry out the verification of several non-
trivial consensus algorithms. We have specified and generated code
for a complete version of Lamport’s Paxos algorithm (verification
of its high-level requirements is on-going, but should be complete
before 2012).

References
[1] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton,
Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computa-
Crecel, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler,
with the Nuprl proof development system. Prentice-Hall, Inc., Upper

For each of the event class combinators \( C \) mentioned above, we
have proved in Nuprl an equivalence relation that expresses when
an element \( v \) is observed by \( C \). We use these lemmas to prove
properties of protocols.

Inductive logical forms. The inductive logical form of a spec-
ification is a first order formula that characterizes completely the
observations (the responses) made by the main class of the specifi-
cation. 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. With an inductive logical form
we can easily prove invariants of the specification by induction on
causal order. For example, fig. 1 presents the inductive logical form
automatically generated for the “ping-pong” specification defined
above. Our “ping-pong” specification can either observe a message
produced by the Handler class (corresponding to the violet part)
and by the ReplyToPing (corresponding to the blue part). If Handler
observes the sending of a message \( m \), it means that a `start` mes-
sage has been received at location \( p \) in the past and that \( m \) is either
a `ping` message sent to one of the locations from the bag locs,
or a `out` message sent in response to a `pong` message.

Conclusion. Our EventML tool provides a language for ele-
egant, abstract specification of distributed algorithms that use asyn-
chronous message passing. It is not yet suited for reasoning about
concurrent shared memory systems.

Programmability and class relation. As mentioned above, event
classes can be seen as having two facets: a programming one and a
logical one. We have proved in Nuprl that all the event classes
mentioned above are programmable, and we have extracted (dis-
 tributed) programs from these proofs. Using these basic results on
programmability, Nuprl can automatically prove that the main class
of an EventML specification is programmable and then extract the
implementation (the set of processes that implement the Message
Automaton) from the proof. The EventML tools can mimic the tac-
tic that Nuprl uses and synthesize code from specifications directly
in EventML, without using Nuprl. If further assurance is needed
Nuprl can check that the synthesized code agrees with the code
extracted from a proof.

Event classes also have a logical aspect. Given an event \( e \) in
some event structure, a class \( X \) of type \( T \), and a element \( v \) in
\( T \), we write \( v \in X(e) \) if \( X \) observes \( v \) at event \( e \), i.e., \( v \)
is in the bag of observations that the class \( X \) makes at event \( e \).
This relation between observed elements, events, and classes is
called the class relation. One can then express the class relation of
combinators in terms of their components. For example, one can
express \( v \in X(X >> Y)(e) \) in terms of \( x \in X(e) \) for observations
\( x \) made by \( X \) and in terms of \( y \in Y(x) \) for observations \( y \)
made by \( Y \).