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

Mark Bickford, Vincent Rahli, Robert Constable

Cornell University and ATC-NY

January 24, 2012
Summary

We have:

- A logical specification language (the logic of events) that formalizes the message sequence diagrams systems engineers use.
- A logical and compositional abstraction (event classes) from which we can synthesize code.
- A language (EventML) for defining event classes and their high-level properties.
- Automated tools that prove invariants and derive “inductive logical forms” that streamline the proofs of distributed algorithms.
  - In two days we now construct proofs of agreement and validity properties of a consensus algorithm.
  - Those proofs used to take a month to create.
Proofs as programs $\rightarrow$ Proofs as processes

- Programs are the evidence for Propositions.
Proofs as programs → Proofs as processes

- Programs are the evidence for Propositions.
- Event ordering = \( \langle E, \text{loc}(e), \text{info}(e), e_1 < e_2 \rangle + \text{six axioms} \)
- Event Logic = propositions in CTT about event orderings
- Evidence ?? could be IO-Automata, \( \pi \)-calculus, ...

\[\begin{align*}
\text{msg}_a & \quad \text{loc}_1 \\
\text{msg}_b & \quad \text{loc}_2 \\
\text{msg}_c & \quad \text{loc}_3 \\
\text{msg}_d & \\
\text{msg}_e & \\
\text{msg}_f & \\
\end{align*}\]
**Event class**: the link to computation

An event class $X$ of type $\text{class}(T)$ is both

- A relation $v \in X(e)$
  - $X$ observes $v$ at event $e$
  - $X$ associates information $v$ with event $e$
- A function $X : EO \rightarrow E \rightarrow \text{Bag}(T)$
An event class \( X \) of type \( \text{class}(T) \) is both

- A relation \( v \in X(e) \)
  - \( X \) observes \( v \) at event \( e \)
  - \( X \) associates information \( v \) with event \( e \)
- A function \( X : EO \rightarrow E \rightarrow \text{Bag}(T) \)
- \( v \in \text{Base}(hdr,\text{type})(e) \Leftrightarrow \text{info}(e) = \langle hdr,\text{type},v \rangle \)
Example: consensus safety properties

**Agreement**

If commands \( c \) and \( c' \) are chosen for the \( n^{th} \) command then \( c = c' \).

\[
\forall e_1,e_2:E. \; \forall n:\mathbb{Z}. \; \forall c,c':\text{Cmd}.
\langle n, c \rangle \in \text{notify'}\text{base}(e_1)
\Rightarrow \langle n, c' \rangle \in \text{notify'}\text{base}(e_2)
\Rightarrow c = c'
\]

**Validity** Any command decided on must have been proposed.

\[
\forall e:E. \; \forall n:\mathbb{Z}. \; \forall c:\text{Cmd}.
\langle n, c \rangle \in \text{notify'}\text{base}(e)
\Rightarrow (\exists e':E. \; (e' < e) \land
\langle n, c \rangle \in \text{propose'}\text{base}(e'))
\]
Event class combinators
(used here to structure 2/3 majority consensus algorithm)

\[
\text{main} = \text{Replica} \circ \text{locs}
\]

\[
\text{Replica} = \text{NewVoters} \ggg \bot \text{p.Voter p}
\]
Event class combinators
(used here to structure 2/3 majority consensus algorithm)

main = Replica @ locs

Replica = NewVoters >>= \ p. Voter p

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

Round (ni,c) = SendVotes (ni,c)
|| Once (Quorum ni)

Event classes and combinators are expressible in EventML.
Computation and logic

Event classes have two facets:

- **computational:**
  - they can be implemented as processes (tail recursive)
  - program for each combinator derived from constituent programs
  - all constructions proved correct in Nuprl
  - result: a verified code synthesizer from event classes to processes

- **logical:**
  - they specify information flow (using the class relation)
  - relation for each combinator derived from constituent relations
  - derived relations proved correct in Nuprl
  - result: a verified translator from event classes to logical relations
Cooperation with a Logical Programming Environment (LPE)
EventML prelude

specification rsc4

(* ———— PARAMETERS ———— *)

(* consensus on commands of arbitrary type Cmd with equality decider *)
parameter Cmd, cmdeq : Type * Cmd Deq

parameter coeff : Int
parameter flrs : Int (* max number of failures *)
parameter locs : Loc Bag (* set of exactly (3 * flrs + 1) locations *)
parameter clients : Loc Bag (* locations of the clients to be notified *)

(* ———— CONSTANTS ———— *)
import length poss—maj list—diff deq—member from—upto Memory—class
int—list—member

(* ———— TYPE FUNCTIONS ———— *)
type Inning = Int
type CmdNum = Int
type CI = CmdNum * Inning
type CC = CmdNum * Cmd
type Vote = (CI * Cmd) * Loc

(* ———— INTERFACE ———— *)
internal vote : Vote
internal retry : CI * Cmd
internal decided : CC
output notify : CC
input propose : CC
(* — inputs — *)
let vote2prop loc (((n, i), c), loc') = {(n, c)} ;;
class Proposal = propose 'base || (vote2prop o vote 'base) ;;

(* — output — *)
let when_new_proposal loc (n, c) (max, missing) =
  if n > max or deq-member (op =) n missing then {(n, c)} else {} ;;

(* — update — *)
let update_replica (n, c) (max, missing) =
  if n > max
  then (n, missing ++ (from-upto (max + 1) n))
  else if deq-member (op =) n missing
  then (max, list-diff (op =) missing [n])
  else (max, missing) ;;

(* — New votes state — *)
class ReplicaState = Memory—class update_replica (init (0, nil)) Proposal ;;

(* — New votes observer — *)
class NewVoters = when_new_proposal o (Proposal, ReplicaState) ;;

(* —— Replica —— *)
class Replica = NewVoters >>= Voter ;;

(* —— Main program —— *)
main Replica @ locs ;;
EventML assertions

\[(\ast \quad \text{state} \quad \ast)\]
\[
\text{class} \ \text{ReplicaState} = \text{Memory}\quad \text{class} \quad \text{update}_\text{replica} \ (\text{init} \ (0, \text{nil})) \ \text{Proposal} \ ; ;
\]

\[(\ast \quad \text{invariants} \quad \ast)\]
\[
\text{invariant} \ \text{replica}_\text{inv} \ \text{on} \ (\text{max}, \text{missing}) \ \text{in} \ \text{ReplicaState} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{max} \geq 0 \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{forall} \ x : \text{Int}, \ \text{int-list-member} \ x \ \text{missing} \ \Rightarrow \ \text{max} \geq x \ \text{forall} \ x > 0 ; ;
\]

Automated tactics prove many assertions automatically.
Inductive logical form (ILF)
automatically generated, automatically proved

∀[Cmd:ValueAllType]. ∀[clients:bag(Id)]. ∀[cmdeq:EqDecider(Cmd)]. ∀[coeff,flrs:Z]. ∀[locs:bag(Id)].
(⟨rcvr, rsc4_vote’msg(Cmd;<<(num, rnd>, c>, sndr)>⟩ ∈ rsc4_Main(e)
⇐⇒ loc(e) ∈ locs
∧ (rcvr ∈ locs ∧ (sndr = loc(e)))
∧ (∃e’:{e’:E| e’ ≤ loc e }
(((∃max:Z
   (∃missing:Z List
     (⟨max, missing⟩ ∈ rsc4_ReplicaState(Cmd)(e’) ∧ ((max < num) ∨ (num ∈ missing))))))
∧ (∃c’:Cmd
   (((e = e’) ∧ (c = c’) ∧ (rnd = 0))
   ∨ ((∃e1:{e1:E| e1 ≤ loc e }
     (((∃maxr:Z. (maxr ∈ rsc4_NewRoundsState(Cmd) num(e1) ∧ (maxr < rnd)))
     ∧ (⟨<num, rnd>, c⟩ ∈ rsc4_retry’base(Cmd)(e1)
     ∨ (∀sndr’:Id. ⟨<num, rnd>, c>, sndr’⟩ ∈ rsc4_vote’base(Cmd)(e1))))
     ∧ (e = e1)))))
∧ (no rsc4_Notify(Cmd;clients) num between e’ and e)))
∧ (⟨num, c’⟩ ∈ rsc4_propose’base(Cmd)(e’)
∨ (∃rnd’:Z. ∃sndr’:Id. ⟨<num, rnd’>, c’, sndr’⟩ ∈ rsc4_vote’base(Cmd)(e’))))))
The right abstractions, embedded in a language that can interface with automated theorem provers gives us the ability to synthesize code that provably satisfies high-level specifications.