Correct-by-constuction, Attack-tolerant Critical systems

Automating Protocol Synthesis

Robert Constable, Robbert van Renesse, Vincent Rahli, Nicolas Schiper, Rich Eaton
– Cornell University
Mark Bickford
– ATC-NY

Correct-by-construction

- We make formal proofs that high-level system requirements are achievable.
- We synthesize system code from the proofs.
- Milestone: We synthesized a fault-tolerant consensus algorithm and deployed it as a component of ShadowDB, a replicated database.

Attack tolerant

- Innate Immunity
  - We prove that the system tolerates certain kinds and numbers of failures (crash, send-omission, etc.) under some assumptions on environment
- Population diversity
  - to thwart attacks not covered by innate immunity, we
  - make variant proofs which synthesize variant algorithms
  - run the synthesized code in variant runtime evaluators (in various languages)
  - (planned) pro-actively reconfigure to use variants in a unpredictable way

Critical components

Empirical observation: There are crucial components in the "stack" of many real-world systems that only a few "gurus" understand and maintain.

Why? In a running system these components have many dynamically changing, loosely coupled parts that achieve their global requirements for subtle reasons.

The Problem:

Such components are difficult to get right in the first place, and cannot be quickly changed if and when a flaw or exploitable feature of their design is discovered.

Our Solution: (Semi-) Automate the reasoning the "guru" uses to understand how the complex component works and why it is correct. Synthesize the code for the component from this reasoning.

Formalizing “guru” reasoning

- Process algebra?   No
- Temporal logic?   Not much
- Refinement maps? Sometimes

- Reason directly about interacting modules/actors/worker threads with input, output, and state.
  - Specify the state change and output for each module.
  - Define and prove local invariants.
  - Prove global invariants.

- I/O Automata?
  - Almost, but we need a better way to reason about properties of dynamically created processes

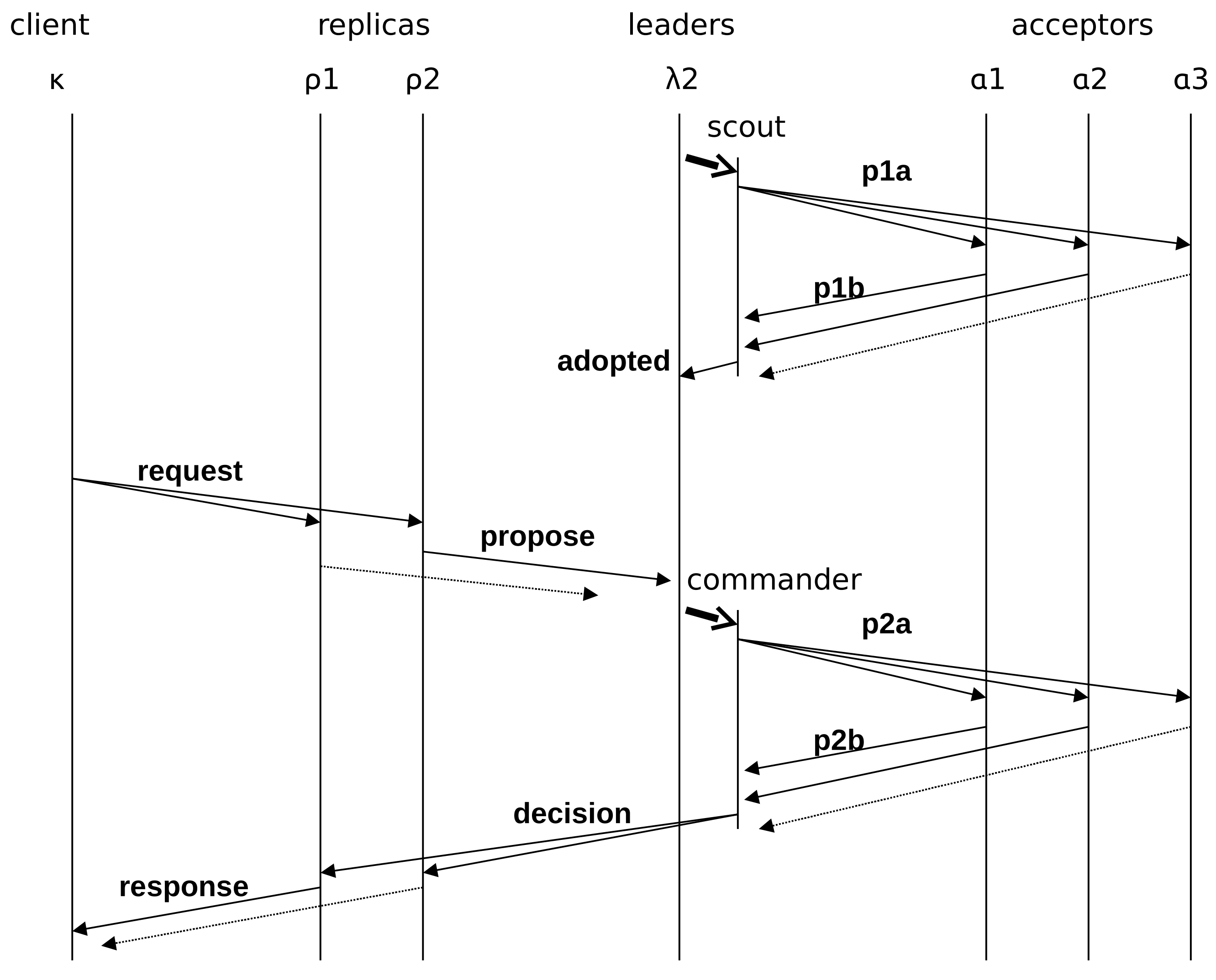
A Thread from Lamport's Paxos consensus algorithm

```
process Scout(λ, acceptors, b)
var waitfor := acceptors, pvalues := ∅;
   ∀α ∈ acceptors : send(α, p1a, self(), b );
  for ever
   switch receive()
     case p1b, α, b' , r :
       if b' = b then
          pvalues := pvalues ∪ r;
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            send(λ, adopted, b, pvalues );
            exit();
         end if;
       else
         send(λ, preempted, b );
         exit();
  end switch;
```

(From an explanation of “multi-decree” Paxos, in Robbert van Renesse's

“Paxos made moderately complex”)

Interacting processes in Paxos



Our Logical Method

- Logic of events = simple formal theory of mathematical structure corresponding to message sequence diagrams

Our Logical Method

- Logic of events = simple formal theory of mathematical structure corresponding to message sequence diagrams
- Processes described abstractly as "event classes" in EventML using "event class combinators", X || Y, F o (X, Y), Prior(X), Once(X),
  - X >>= Y *delegation combinator* expresses dynamic process creation (classes form a monad)

EventML automatically synthesizes code (a set of process terms that execute in a message passing evaluator)

EventML automatically generates a logical form.

Nuprl then generates and proves a simplified "inductive logical form" (ILF)

EventML is both a programming and specification language.

Our Logical Method

- Logic of events = simple formal theory of mathematical structure corresponding to message sequence diagrams
- Processes described abstractly as “event classes” in EventML using “event class combinators”, X || Y, F o (X, Y), Prior(X), Once(X),
  - X >>= Y *delegation combinator* expresses dynamic process creation (classes form a monad)
- EventML automatically synthesizes code (interpreted by a message passing evaluator)

Our Logical Method

- Logic of events = simple formal theory of mathematical structure corresponding to message sequence diagrams
- Processes described abstractly as “event classes” in EventML using “event class combinators”, X || Y, F o (X, Y), Prior(X), Once(X),
  - X >>= Y *delegation combinator* expresses dynamic process creation (classes form a monad)
- EventML automatically synthesizes code (interpreted by a message passing evaluator)
- EventML automatically generates a logical meaning.
  - Nuprl then generates and proves a simplified “inductive logical form” (ILF)

Our Logical Method

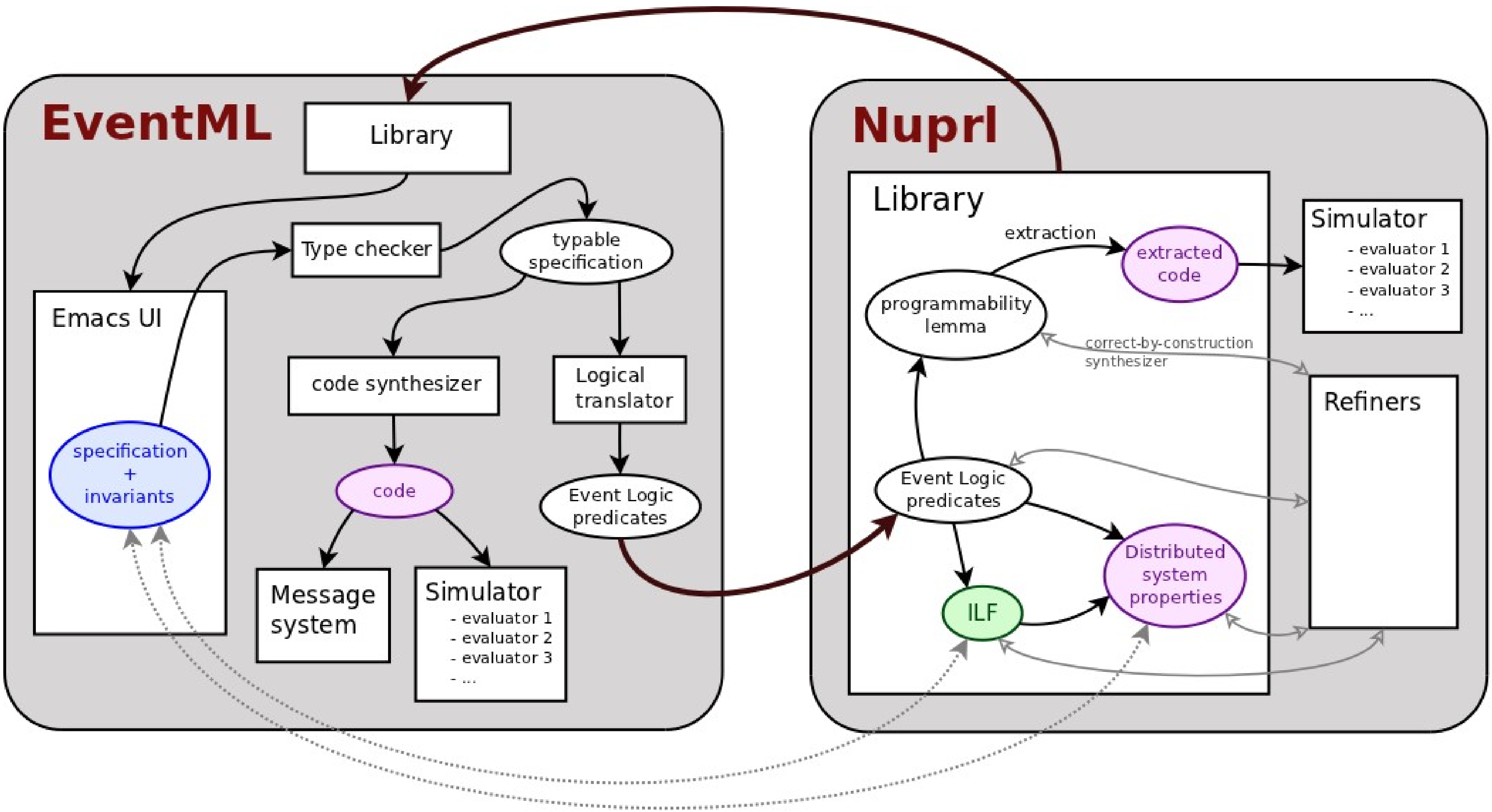
- Logic of events = simple formal theory of mathematical structure corresponding to message sequence diagrams
- Processes described abstractly as “event classes” in EventML using “event class combinators”, X || Y, F o (X, Y), Prior(X), Once(X),
  - X >>= Y *delegation combinator* expresses dynamic process creation (classes form a monad)
- EventML automatically synthesizes code (a set of process terms that execute in a message passing evaluator)
- EventML automatically generates a logical form.
  - Nuprl then generates and proves a simplified “inductive logical form” (ILF)
- EventML is both a programming and specification language.

Progress since last PI meeting

- Many enhancements to EventML
  - Abstract data types
  - Invariant assertions, ordering properties
  - classrec R p = X p || Y p >>= R
- Generation and simplification of ILF
  - Using domain specific reasoners
  - Rewriting, quantifier elimination, etc. all proved by Nuprl tactics.
- Synthesized code deployed
  - Several versions of evaluators working
  - Consensus code being used in replicated database (ShadowDB).

Synthesized consensus protocols

- 3f+1 “simple” consensus algorithm
  - Written in EventML with assertions
  - Most local invariants automatically proved
  - Using automatically generated ILF we proved the global consistency & validity properties in about two days (previous effort took two months)
  - Synthesized code is running in reconfiguration service of ShadowDB
- Paxos nearly finished



EventML (built by Vincent Rahli) cooperates with Nuprl at every stage of program development.

A Thread from Lamport's Paxos consensus algorithm

```
 process Scout(λ, acceptors, b)
var waitfor := acceptors, pvalues := ∅;
   ∀α ∈ acceptors : send(α, p1a, self(), b );
  for ever
   switch receive()
      case p1b, α, b' , r :
       if b' = b then
           pvalues := pvalues ∪ r;
           waitfor := waitfor − {α};
           if |waitfor| < |acceptors|/2 then
             send(λ, adopted, b, pvalues );
             exit();
          end if;
      else
        send(λ, preempted, b );
        exit();
  end switch;
```

(From an explanation of “multi-decree” Paxos, in Robbert van Renesse's

“Paxos made moderately complex”)

Part of EventML for Paxos

```
class ScoutNotify b = Output(\ldr.p1a'broadcast accpts (ldr, b));;

let on_p1b bnum loc (acloc,(b',pvals)) (waitfor,pvalues) =
  if eq_bnums bnum b'
  then let waitfor' = bag-remove (op =) waitfor acloc in
       let pvalues' = append_news same_pvalue pvalues pvals in
         (waitfor',pvalues')
  else (waitfor,pvalues) ;;

class ScoutState b = State1 (\loc.init_scout) (on_p1b b) p1b'base;;

let scout_output b ldr (a,(b',r)) (waitfor,pvalues) =
  if eq_bnums b b'
  then if bag-size waitfor < threshold
       then { adopted'send ldr (b,pvalues) }
       else {}
  else { preempted'send ldr b' };;

class ScoutOutput b = Once((scout_output b) o (p1b'base, ScoutState b));;

class Scout b = ScoutNotify b || ScoutOutput b ;;
```

Part of the Inductive Logical Form (ILF) for Paxos

```
(∀[bnum:BNum]. ∀[accpts:bag(Id)]. ∀[Op,Cid:{T:Type| valueall-type(T)} ].
 ∀[eq_Cid:EqDecider(Cid)].∀[es:EO’]. ∀[e:E]. ∀[i:Id]. ∀[m:Message].
   {<i, m> ∈ paxos_scout_output(Cid;Op;accpts) bnum@Loc o (Loc,
            paxos_p1b’base(Cid;Op), paxos_ScoutState(Cid;Op;accpts;eq_Cid) bnum)(e)
   ⟺ ↓(header(e) = ‘‘paxos p1b‘‘)
        ∧ (type(info(e)) = (Id × BNum × ((BNum × ℤ × Id × Cid × Op) List)))
        ∧ (i = loc(e))
        ∧ (((bnum = (fst(snd(body(info(e))))))
          ∧ (bag-size(fst(State of Scout bnum at e)) < paxos_threshold(accpts))
          ∧ (m = make-Msg(‘‘paxos adopted‘‘;
                          BNum × ((BNum × ℤ × Id × Cid × Op) List);
                          <bnum , snd(State of Scout (for bnum) at e)>)))
          ∨ ((¬(bnum = (fst(snd(body(info(e)))))))
            ∧ (m = make-Msg(‘‘paxos preempted‘‘;BNum;fst(snd(body(info(e))))))))})
```

The ILF is readable, and usually more informative than the EventML or pseudo-code. It is used automatically by our tactics to prove global properties of the algorithm.

Part of the Inductive Logical Form (ILF) for Paxos

```
(∀[bnum:BNum]. ∀[accpts:bag(Id)]. ∀[Op,Cid:{T:Type| valueall-type(T)} ].
 ∀[eq_Cid:EqDecider(Cid)].∀[es:EO']. ∀[e:E]. ∀[i:Id]. ∀[m:Message].
   {<i, m> ∈ paxos_scout_output(Cid;Op;accpts) bnum@Loc o (Loc,
             paxos_p1b'base(Cid;Op), paxos_ScoutState(Cid;Op;accpts;eq_Cid) bnum)(e)
   ⟺ ↓(header(e) = ``paxos p1b``)
       ∧ (type(info(e)) = (Id × BNum × ((BNum × ℤ × Id × Cid × Op) List)))
       ∧ (i = loc(e))
       ∧ (((bnum = (fst(snd(body(info(e))))))
         ∧ (bag-size(fst(State of Scout bnum at e)) < paxos_threshold(accpts))
         ∧ (m = make-Msg(``paxos adopted``;
                         BNum × ((BNum × ℤ × Id × Cid × Op) List);
                         <bnum , snd(State of Scout (for bnum) at e)>)))
         ∨ ((¬(bnum = (fst(snd(body(info(e)))))))
           ∧ (m = make-Msg(``paxos preempted``;BNum;fst(snd(body(info(e)))))))})
```

The ILF is readable, and usually more informative than the EventML or pseudo-code. It is used automatically by our tactics to prove global properties of the algorithm.

Part of the Inductive Logical Form (ILF) for Paxos

```
(∀[bnum:BNum]. ∀[accpts:bag(Id)]. ∀[Op,Cid:{T:Type| valueall-type(T)} ].
 ∀[eq_Cid:EqDecider(Cid)].∀[es:EO’]. ∀[e:E]. ∀[i:Id]. ∀[m:Message].
   {<i, m> ∈ paxos_scout_output(Cid;Op;accpts) bnum@Loc o (Loc,
             paxos_p1b’base(Cid;Op), paxos_ScoutState(Cid;Op;accpts;eq_Cid) bnum)(e)
   ⟺ ↓(header(e) = ‘‘paxos p1b‘‘)
        ∧ (type(info(e)) = (Id × BNum × ((BNum × ℤ × Id × Cid × Op) List)))
        ∧ (i = loc(e))
        ∧ (((bnum = (fst(snd(body(info(e))))))
          ∧ (bag-size(fst(State of Scout bnum at e)) < paxos_threshold(accpts))
          ∧ (m = make-Msg(‘‘paxos adopted‘‘;
                         BNum × ((BNum × ℤ × Id × Cid × Op) List);
                         <bnum , snd(State of Scout (for bnum) at e)>)))
          ∨ ((¬(bnum = (fst(snd(body(info(e)))))))
            ∧ (m = make-Msg(‘‘paxos preempted‘‘;BNum;fst(snd(body(info(e))))))))})
```

The ILF is readable, and usually more informative than the EventML or pseudo-code. It is used automatically by our tactics to prove global properties of the algorithm.

Part of the Inductive Logical Form (ILF) for Paxos

```
(∀[bnum:BNum]. ∀[accpts:bag(Id)]. ∀[Op,Cid:{T:Type| valueall-type(T)} ].
 ∀[eq_Cid:EqDecider(Cid)].∀[es:EO’]. ∀[e:E]. ∀[i:Id]. ∀[m:Message].
   {<i, m> ∈ paxos_scout_output(Cid;Op;accpts) bnum@Loc o (Loc,
            paxos_p1b’base(Cid;Op), paxos_ScoutState(Cid;Op;accpts;eq_Cid) bnum)(e)
   ⟺ ↓(header(e) = ‘‘paxos p1b‘‘)
       ∧ (type(info(e)) = (Id × BNum × ((BNum × ℤ × Id × Cid × Op) List)))
       ∧ (i = loc(e))
       ∧ (((bnum = (fst(snd(body(info(e))))))
         ∧ (bag-size(fst(State of Scout bnum at e)) < paxos_threshold(accpts))
         ∧ (m = make-Msg(‘‘paxos adopted‘‘;
                         BNum × ((BNum × ℤ × Id × Cid × Op) List);
                         <bnum , snd(State of Scout (for bnum) at e)>)))
         ∨ ((¬(bnum = (fst(snd(body(info(e)))))))
           ∧ (m = make-Msg(‘‘paxos preempted‘‘;BNum;fst(snd(body(info(e))))))))})
```

The ILF is readable, and usually more informative than the EventML or pseudo-code. It is used automatically by our tactics to prove global properties of the algorithm.

Summary/ Next steps

- Synthesis of complex distributed algorithms from proofs works
  - Abstractions, automation essential
- Next steps
  - More variants of more protocols
  - Reason about capabilities/tags so that we can synthesize code that uses more CRASH technology