

Correct-by-Construction Methodology

CRASH Methodology for Correct-by-construction Attack-tolerant Systems

Also see the: [CRASH Project Home](#) | [About CRASH](#) | [Software](#) | [People](#) | [Publications](#)

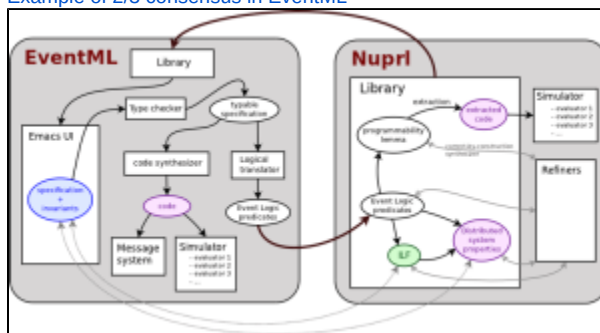
Background

Our method uses formal proofs of high-level system requirements then to synthesize code from the proven specification. Within our work on distributed systems we use the following concepts:

- Logic of Events: a simple formal theory of mathematical structure corresponding to message sequence diagrams
 - [Logic of Events presentation at LADA 2012](#) by Robert Constable, Mark Bickford, and Vincent Rahli. 2012
 - [Automated Proof of Authentication Protocols in a Logic of Events](#) by Mark Bickford. 2010
 - [A Causal Logic of Events in Formalized Computational Type Theory](#) by Mark Bickford, Robert L. Constable. 2005
- Event Classes: abstract description of processes
 - [Investigating Correct-by-Construction Attack-Tolerant Systems](#) by Robert Constable, Mark Bickford, Robbert Van Renesse. 2010
 - [Generating Event Logics with Higher-Order Processes as Realizers](#) by Mark Bickford, Robert L. Constable, David Guaspari. 2010
 - [Component Specification Using Event Classes](#) by Mark Bickford. 2009

The main tool we use is [EventML](#) which is a programming and specification language. EventML, built by [Vincent Rahli](#), cooperates with the Nuprl interactive theorem prover at every stage of program development to help programmers ensure correctness, document the code, and support modifications and improvements. It generates an *Inductive Logical Form* that proves the specification and can also automatically synthesize code.

- Resources for EventML
 - EventML documentation and downloads available at <http://www.nuprl.org/software/>
 - [EventML Tutorial](#)
 - [Example of 2/3 consensus in EventML](#)



Code *diversity* is created during the process. We can introduce variants at the EventML specification and code synthesis.

Methodology

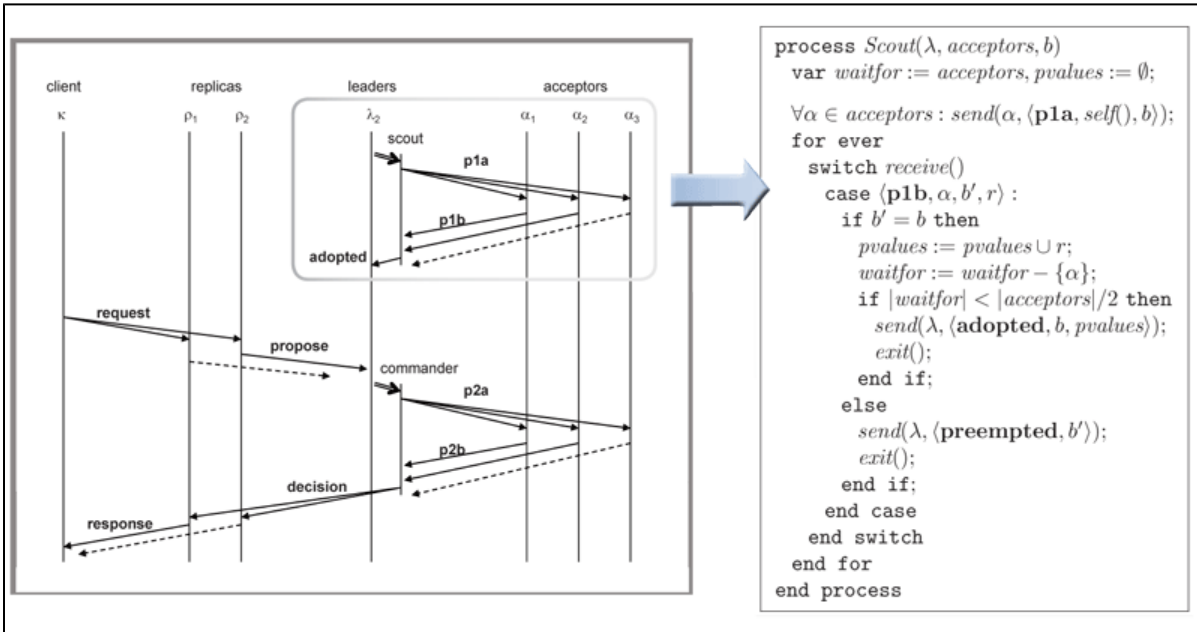
To create correct-by-construction code we:

1. Write the specification in [EventML](#)
2. Automatically generate and prove an Inductive Logical Form of the specification
3. Synthesize code
4. Diversify and deploy code

For more details about the example below see Mark's [presentation at the May 2012 CRASH meeting](#).

Example with Consensus

In this example we will look at Paxos consensus, focusing on specifying the Scout protocol.



Step 1 - We write the Scout specification in EventML

EventML uses classes from the Formal Digital Library to describe events and protocols.

```

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 bnum 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 ;;

```

Step 2 - Generate the Inductive Logical Form (ILF) from the Scout specification

Here EventML interfaces with Nuprl's formal library and theorem prover. Using the distributed prover, we generate a proof of the protocol which results in a readable Inductive Logical Form. If there are any issues with the proof we revise the specification and reiterate the process with the ILF.

Inductive Logical Form proof of specification

```

[ $\forall$  [bnum:BNum].  $\forall$  [accpts:bag(Id)].  $\forall$  [Op,Cid:{T:Type | valueall-type(T)}].
 $\forall$  [eq_Cid:EqDecider(Cid)].  $\forall$  [es:EO'].  $\forall$  [e:E].  $\forall$  [i:Id].  $\forall$  [m:Message].
{<i, m>  $\in$  paxos_scout_output(Cid;Op;accpts) bnum@Loc o (Loc,
    paxos_plb'base(Cid;Op), paxos_ScoutState(Cid;Op;accpts;eq_Cid) bnum)(e)}
 $\iff$   $\downarrow$  (header(e) = "paxos plb")
 $\wedge$  (type(info(e)) = (Id  $\times$  BNum  $\times$  ((BNum  $\times$   $\mathbb{Z}$   $\times$  Id  $\times$  Cid  $\times$  Op) List)))
 $\wedge$  (i = loc(e))
 $\wedge$  ((bnum = (fst(snd(body(info(e)))))
 $\wedge$  (bag-size(fst(State of Scout bnum at e)) < paxos_threshold(accpts))
 $\wedge$  (m = make-Msg("paxos adopted";
    BNum  $\times$  ((BNum  $\times$   $\mathbb{Z}$   $\times$  Id  $\times$  Cid  $\times$  Op) List);
    <bnum, snd(State of Scout (for bnum) at e)>)))
 $\vee$  (( $\neg$ (bnum = (fst(snd(body(info(e)))))
 $\wedge$  (m = make-Msg("paxos preempted";BNum;fst(snd(body(info(e)))))
```

A scout is a subprocess that tries to get a ballot number adopted by a majority of acceptors. A scout, working on ballot number **bnum**, sends a message **m** to location **i** if and only if the following happens:

1. The scout reacts to the receipt of **p1b** message (from an acceptor).
2. The **p1b** message contains a location, ballot number, and proposal list.
3. The scout sends message **m** to itself.
4. If a majority of acceptors have adopted the ballot number **bnum**, then message **m** is an **adopted** message.
5. However, if the scout receives a **p1b** message that contains a ballot number different from the one it is currently working on, the scout sends a **preempted** message.

Step 3 - Synthesize Consensus Code

Using EventML's correct-by-construction synthesizer we generate code from the specification.

Step 4 - Diversify & Deploy

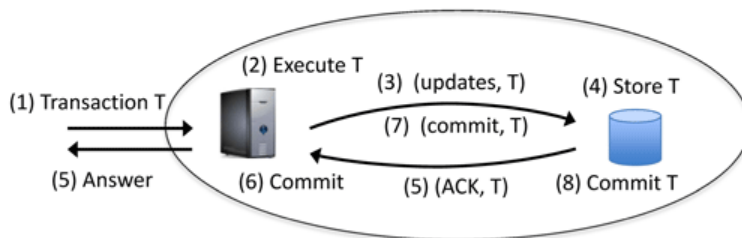
Engaging our diversity in classes and state machines we generate multiple verified versions of the code. Then the final step before deployment is to test the code in the EventML simulator. The result is a correct-by-construction synthesized version of Paxos with multiple code variants.

Example Deployment: ShadowDB

ShadowDB is a replicated database, created by [Nicolas Schiper](#), on top of a synthesized consensus core. The primary defines the order in which transaction updates are applied. When crashes occur, consensus is used to reconfigure the set of replicas and agree on a prefix of executed transactions.

ShadowDB: A replicated database on top of a synthesized consensus core

There are up to $f = 1$ failures



The primary defines the order in which transactions updates are applied.

At (5) it is safe to answer the client: $f+1$ replicas store the transaction.



When a crash occurs, the next set of replicas is agreed upon using consensus.

Correct-by-construction consensus in EventML

