Interfacing with Proof Assistants for Domain Specific Programming Using EventML

Vincent Rahli

PRL team - Cornell University

July 13, 2012

Credits

- Mark Bickford
- Robert Constable
- David Guaspari
- Richard Eaton
- Vincent Rahli
- Robbert Van Renesse
- Nicolas Schiper
- Jason Wu

Problem

Problem: unverified protocols are wrong.

Goal: automatic synthesis of verified diversifiable distributed systems.

Our solution: building tools that cooperate with a Logical Programming Environment (e.g., a constructive theorem prover).

EventML: specification and programming language

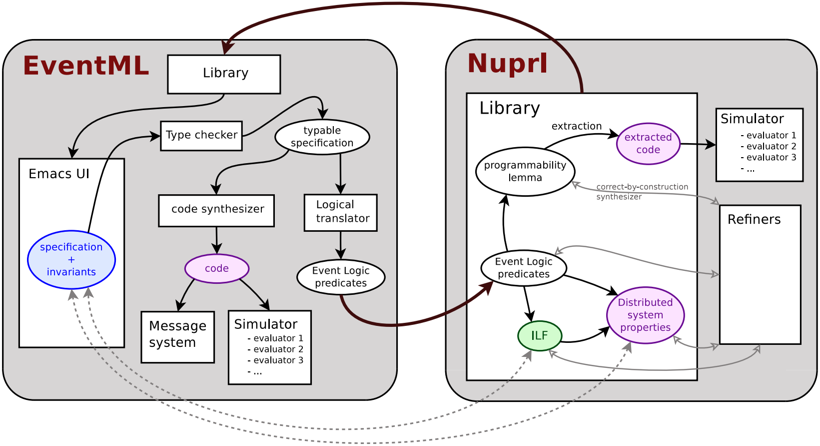
- A ML-like **functional** programming language.
- Features logical constructs (**Logic of Events** combinators).
- To **specify/code** distributed protocols.
- EventML **translates** specifications into event classes.

  Logical aspect

- EventML **synthesizes** distributed programs (in the model underlying the Logic of Events) from specifications.

  Computational aspect

Cooperation with a Logical Programming Environment



Accomplishments

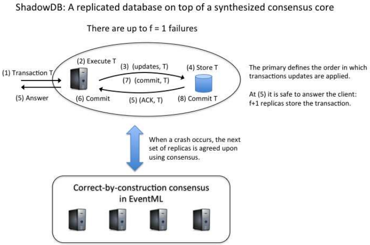
We have specified many distributed protocols.
We have proved the correctness of the following protocols:

- Leader election in a ring.
- Two-thirds consensus protocol.
- Paxos (in progress).

The methodology works!

Nicolas Schiper
(Cornell postdoc) has implemented
a replicated database (ShadowDB)
on top of our synthesized
two-thirds consensus protocol.

It is used!



An example: Maximum using Memory

We have defined state machines in the Logic of Events.

E.g., Memory1.

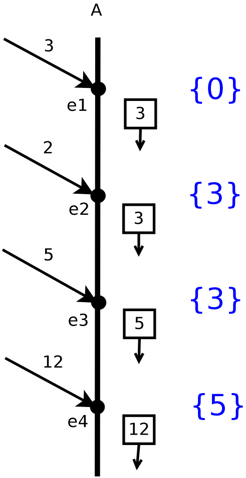
We have automated some reasoning on state machines.

Maximum

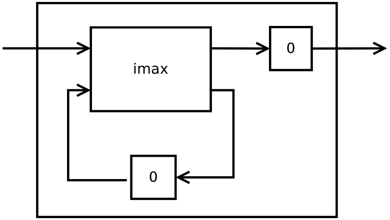
```
input int : Int

class Maximum =
  Memory1 (\loc.{0})
          (\loc.\x.\s. imax x s)
          int'base
;;
```

Intuition: at any event, computes the maximum of the integers received in the past.

Maximum

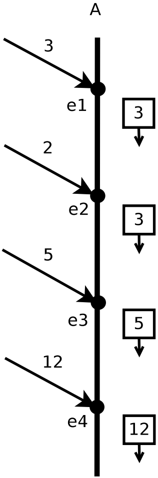



```
class Maximum =
  Memory1 (\loc.{0})
               (\loc.\x.\n.imax x n)
               int'base
```



Maximum

```
class Obs1 =
  let F loc x n = if n > 3 & n < 20
                  then {imax x n}
                  else {}
  in F o (int'base,Maximum)
```



{} {}

{} {}

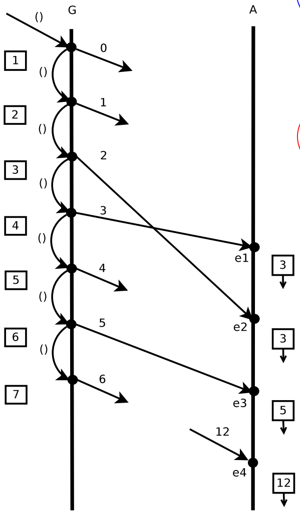
{} {(3,5)}

{12} {(5,12)}

```
class Obs2 =
  let F loc x n = if n >= 3 & x >= 3
                  then {(n,x)}
                  else {}
  in F o (int'base,Maximum)
```

Maximum

```
input    start : Unit
internal inc   : Unit

class Input = start'base
           || inc'base

class IncState =
  Memory1 (\loc.{0})
          (\loc.\().\n.n+1)
          Input

Class Increment =
  let F loc () n =
    { int'send A n
    ; inc'send loc ()
    }
  in F o (Input,IncState)
```




```
class Obs1 =
  let F loc x n = if n > 3 & n < 20
                  then {imax x n}
                  else {}
  in F o (int'base,Maximum)
```

```
class Obs2 =
  let F loc x n = if n >= 3 & x >= 3
                  then {(n,x)}
                  else {}
  in F o (int'base,Maximum)
```

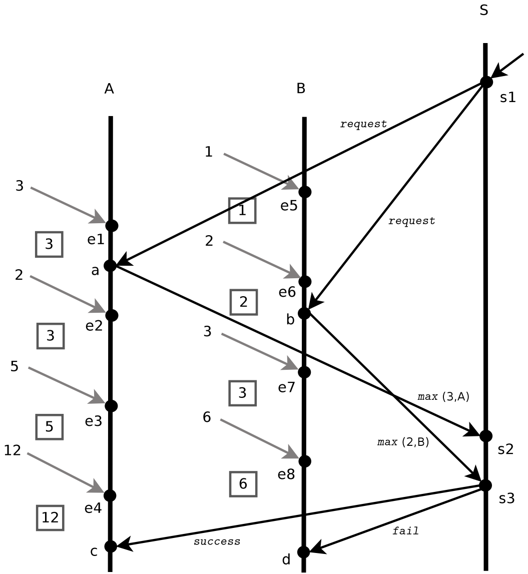
{} {}

{} {}

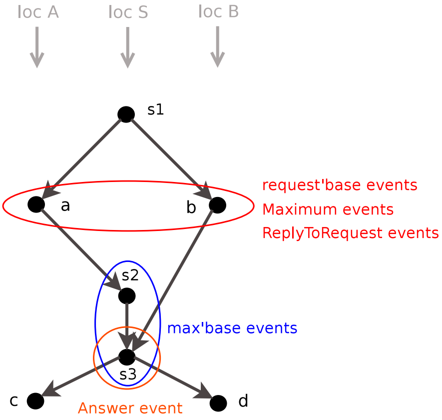
{} {(3,5)}

{12} {(5,12)}

Maximum


S
A
B
s1
request
1
3
e5
1
request
e1
3
2
a
2
e6
2
b
e2
3
3
5
e7
3
max (3,A)
6
e3
5
s2
max (2,B)
12
e8
6
s3
e4
12
fail
success
c
d

Maximum



loc A
loc S
loc B
s1
a
b
request'base events
Maximum events
ReplyToRequest events
s2
max'base events
s3
c
d
Answer event

Maximum

One can specify state machine invariants in EventML:

```
invariant pos_max on n in Maximum
  == n >= 0;;

progress inc_max on n1 then n2 in Maximum
   with n in int'base and s => n > s
  == n2 > n1;;

memory mem_max on n1 then n2 in Maximum
   with n in int'base
  == n2 >= n /\ n2 >= n1;;
```

Nuprl automatically proves these invariants.

What's next?

- Automation.
- Correct-by-construction optimizations.
- More expressive types: refinement types, dependent types...