Aneris: A Diversified and Correct-by-Construction Broadcast Service

Vincent Rahli, Nicolas Schiper, Robbert Van Renesse,
Mark Bickford, and Robert L. Constable
rahli@cs.cornell.edu
www.nuprl.org/Publications

October 30, 2012

Table of contents

Goals

Long term goal: Platform to develop provably correct programs.

Problem: Distributed programs are hard to implement, even more so if they have to be fault-tolerant.

Goal: Build Aneris: a synthesized and verified ordered broadcast service with diversity.

Table of Contents

Aneris

A synthesized and verified ordered broadcast service with diversity.

An ordered broadcast service?

➲ A fault-tolerant service using **state machine replication**.

➲ The service can still be used even when machines crash (up to a certain number of failures).

➲ The service receives requests from clients and ensures that they will be delivered by the replicas in the **same order**.

➲ Ordered delivery is implemented using consensus on the i^{th} command, and this for every i.

Aneris

A synthesized and verified ordered broadcast service with diversity.

An ordered broadcast service?

➲ Each replica fills a sequence of slots with requests from clients.

➲ Once a replica has filled a slot s with a request r, it delivers a message (r, s) to the clients.

Aneris ensures:

(1) *validity*: each delivery is initiated by a request.

(2) *uniqueness*: a replica delivers a given message at most once.

Aneris

A synthesized and verified ordered broadcast service with diversity.

An ordered broadcast service?

(3) *agreement*: for any slot s, if $(r1, s)$ and $(r2, s)$ get delivered then $r1 = r2$.

(4) *termination*: if a replica never crashes, a request r eventually results in a delivery (r, s).

(5) *relay*: if a replica delivers (r, s), then each replica that never crashes eventually delivers (r, s).

(6) *gap-free*: if a replica delivers $(r, s > 0)$ then it has previously delivered $(r', s - 1)$.

Aneris

A synthesized and verified ordered broadcast service with diversity.

Synthesis?

➲ Automatic generation of “code” from “constructive” specifications.

➲ Easier to maintain, modify, and reason about (reasoning is done at the specification level).

Aneris

A synthesized and verified ordered broadcast service with diversity.

Verified?

- Proofs that the specification is correct (w.r.t. some criteria) using a proof assistant (Nuprl [CAB+86, Kre02, ABC+06]).
- Proof that the synthesized code satisfies the specification.
- Some automation.
- One gets **provably correct** code (correct-by-construction).

Aneris

A synthesized and verified ordered broadcast service with diversity.

Diversity?

➲ Diversified for failure independence.

➲ If all the replicas were to run the same code they would share the same vulnerabilities.
All the replicas could crash because of a single bug.

➲ Diversity in space: the replicas run different code.

➲ Still, the replicas may have vulnerabilities that adversaries may try to exploit.

➲ Diversity in time: the code changes over time.

Table of Contents

Diversity

Diversity in space: data structures, evaluation...

Diversity in time: currently Aneris uses 2 consensus protocols (f is the number of tolerated failures):

- $2/3$ consensus:
  - $3f + 1$ replicas ($3f + 1$ machines)
  - At best a single message round
- Paxos Synod:
  - $2f + 1$ acceptors and $f + 1$ leaders (at least $2f + 1$ machines)
  - At best 2 message rounds

Diversity

An attack scenario:

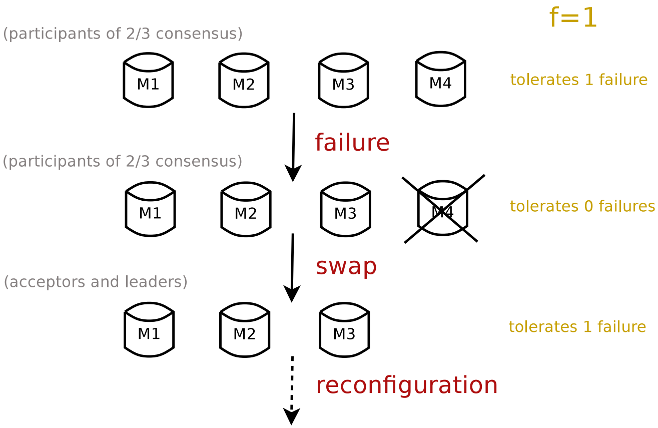
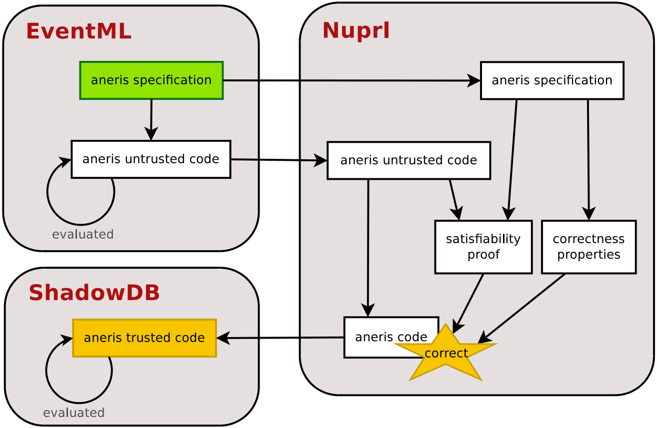


Table of Contents

Programming with the help of a proof assistant

In Nuprl, specifications are expressed in the Logic of Events [Bic09, BC08] (logical framework to reason about and synthesize distributed protocols).



EventML

2/3:

```
..
class TT_Replica = NewVoters >>= Voter;;
main TT_Replica @ locs
```

Paxos Synod:

```
...
class Leader = SpawnFirstScout
            || ((LeaderPropose || LeaderAdopted) >>= Commander)
            || (LeaderPreempted >>= Scout) ;;
main Leader @ ldrs || Acceptor @ accpts
```

Aneris replicas:

```
...
class ReplicaState =
  State(\_.(init_state,{}),
        out_tr propose_inl, swap'base,
        out_tr propose_inr, bcast'base,
        out_tr on_decision, decision'base);;
class Replica = (\_.snd) o ReplicaState ;;
main Replica @ reps
```

Code synthesis

For each combinator of the Logic of Events, we have defined a process (in our General Process Model [BCG10] defined in Nuprl) that implements it.

Most of them are simple recursive functions.

EventML synthesize code and Nuprl (recursively) checks that the code implements the specification.

Code synthesis

Optimized version of the Aneris process:

```
aneris_main-program-opt(Cid;Op;clients;eq_Cid;pax_procs;reps;tt_procs) ==
  λi.case bag-deq-member(λa,b.if a=2 b then inl · else (inr · );i;reps)
     of inl() =>
      fix((λmk-hdf,s.
            (inl (λv.let x,y = v
                     in case name_eq(x;[swap]) ∧b ...
                        of inl(x1) =>
                         let v1 ⟵ ... aneris_propose_inl(Cid;Op;...;...;...;...;...) ...
                         in let x,y = v1 in let v2 ⟵ y @ [] in <mk-hdf <x, y>, v2>
                         | inr(y1) =>
                         case name_eq(x;[bcast]) ∧b ...
                         of inl(x1) =>
                          let v1 ⟵ ... aneris_propose_inr(Cid;Op;...;...;...;...;...) ...
                          in let x,y = v1 in let v2 ⟵ y @ [] in <mk-hdf <x, y>, v2>
                          | inr(y1) =>
                          case name_eq(x;[decision]) ∧b ...
                          of inl(x1) =>
                           let v1 ⟵ ... aneris_on_decision(Cid;Op;...;...;...;...;...;...;...) ...
                           in let x,y = v1 in let v2 ⟵ y @ [] in <mk-hdf <x, y>, v2>
                           | inr(y1) =>
                           let v1 ⟵ s
                           in let x,y = v1 in let v2 ⟵ y @ [] in <mk-hdf <x, y>, v2>) )))
      <aneris_init_state(Cid;Op), []>
      | inr() =>
      inr ·
```

The dots correspond to small terms.

Verification

Using the tools we have built in Nuprl, it took us:

- about 2 days to prove the safety properties of 2/3,
- about 2 weeks for Paxos Synod,
- about 1 additional week to prove full Paxos (Synod + learners),
- a few hours to prove validity.
- Proving the other properties should take us a few more days worth of work.

Verification

We use causal induction and inductive logical forms. Logical explanation of why decisions are made by Paxos:

```
∀[Cmd:{T:Type| valueall-type(T)} ]. ∀[accpts,ldrs:bag(Id)]. ∀[ldrs_uid:Id ⟶ ℤ]. ∀[reps:bag(Id)].
∀[es:EO']. ∀[e:E]. ∀[i:Id]. ∀[p:Proposal].
 (decision'send(Cmd) i p ∈ pax_mb_main(Cmd;accpts;ldrs;ldrs_uid;reps)(e)     decision of p sent to i at e
 ⟺  loc(e) ∈ ldrs                                                        e happens at a leader location
    ∧ (header(e) = ``pax_mb p2b``)
    ∧ (msgtype(e) = P2b)                                          the decision is triggered by a p2b message
    ∧ i ∈ reps                                         the recipient of the decision message is a replica
    ∧ (∃e':{e':E| e' ≤loc e }
       ∃z:PValue                                                    proposal p is extracted from a pvalue z
        ((((header(e') = [propose])                  either pvalue z is made from a proposal and current ballot
         ∧ (msgtype(e') = Proposal)
         ∧ ((↑ (proposal_slot (proposal_cmd LeaderStateFun(e'))))
           ∧ (¬↑ (in_domain (proposal_slot msgval(e')) (proposal_cmd (proposal_cmd LeaderStateFun(e'))))))
         ∧ (z = (mk_pvalue (proposal_slot LeaderStateFun(e')) msgval(e'))))
         ∨ ((header(e') = ``pax_mb adopted``)    or either pvalue z received in an adopted message or in leader state
          ∧ (msgtype(e') = pax_mb_AState(Cmd))
          ∧ ((astate_ballot msgval(e')) = (proposal_slot LeaderStateFun(e')))
          ∧ z ∈ map(λsp.(mk_pvalue (astate_ballot msgval(e')) sp);
                    update_proposals (proposal_cmd (proposal_cmd LeaderStateFun(e')))
                                     (pmax(ldrs_uid) (astate_pvals msgval(e'))))))
        ∧ (no commander_output(accpts;reps) z@Loc            this decision is the first output of the commander
             o (Loc,p2b'base(), CommanderState(accpts) (pval_ballot z) (proposal_slot (pval_proposal z)))
          between e' and e)
        ∧ ((pval_ballot z) = (bl_ballot (p2b_bl msgval(e))))
        ∧ ((proposal_slot (pval_proposal z)) = (p2b_slot msgval(e)))
        ∧ ((pval_ballot z) = (p2b_ballot msgval(e)))    the acceptor that sent the p2b message has accepted pvalue z
        ∧ (#(CommanderStateFun(pval_ballot z;proposal_slot (pval_proposal z);es.e';e)) < threshold(accpts))
        ∧ (p = (pval_proposal z)))))             the commander has received a p2b messages from a majority of acceptors
```

Verification

Tracing back the information flow of the system from outputs to inputs and state variables, we easily proved validity:

```
∀[Cid,Op:ValueAllType].∀[eq_Cid:EqDecider(Cid)].∀[eq_Op:EqDecider(Op)].
∀[accpts,ldrs,locs,reps:bag(Id)].
∀[ldrs_uid:Id ⟶ ℤ]. ∀[flrs:ℤ]. ∀[es:EO'].
            (∀i:Id. ∀s:ℤ. ∀k:Cid. ∀c:Op.
               if Aneris_main() outputs (Aneris_deliver'send() i <s, k, c>)
               then Aneris_broadcast'base() observed <i, k, c>)
  supposing ((∀i:Id. ∀s:ℤ. ∀c:Cid × (Atom List) + (Id × Cid × Op).
               if c23_main() outputs (c23_notify'send([decision]) i <s, c>)
               then c23_propose'base([tt_propose]) observed <s, c>)
        and  (∀i:Id. ∀s:ℤ. ∀c:Cid × (Atom List) + (Id × Cid × Op).
               if cpax_main() outputs (cpax_decision'send([decision]) i <s, c>)
               then cpax_propose'base([pax_propose]) observed <s, c>)
        and  Aneris_message-constraint-p1(es))
```

Verification

That was possible thanks:

- to Nuprl's large library of definitions and facts,
- to the powerful **logic of events** theory developed in Nuprl by Mark Bickford and Robert Constable over the past few years (especially to the **delegation** combinator), and
- to the collaboration between the PRL and system groups at Cornell.

Table of Contents

Current and future work

- Performance
  - We are currently working on formally optimizing the synthesized code in Nuprl.
  - We plan on implementing interpreters and a compiler.
- ShadowDB (implemented by Nicolas Schiper)
  - Replicated database that uses Aneris to handle failures.
  - We plan to replace more of ShadowDB's components by synthesized versions (e.g., reconfiguration module).
  - Designing/running experiments.

Summary

- Synthesized and partially verified an ordered broadcast service called Aneris.
- Diversity in time (protocol swapping). Diversity in space (data structures, evaluators, parameters, . . .).
- Aneris in used by the replicated database ShadowDB that itself will be used by Nuprl.
- **Example that our methodology to specify (using small human manageable components) and verify (ILFs + causal induction) protocols works.**
- Started engaging proof assistants in the programming process using EventML and Nuprl (long term goal).

References I

Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran.
Innovations in computational type theory using Nuprl.
J. Applied Logic, 4(4):428–469, 2006.

Mark Bickford and Robert L. Constable.
Formal foundations of computer security.
In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

Mark Bickford, Robert Constable, and David Guaspari.
Generating event logics with higher-order processes as realizers.
Technical report, Cornell University, 2010.

Mark Bickford.
Component specification using event classes.
In *Component-Based Software Engineering, 12th Int'l Symp.*, volume 5582 of *LNCS*, pages 140–155. Springer, 2009.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith.
Implementing mathematics with the Nuprl proof development system.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

Christoph Kreitz.
The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide.
Cornell University, Ithaca, NY, 2002.
`www.nuprl.org/html/02cucs-NuprlManual.pdf`.