

AFFIRM Year 2

August 9, 2016 - NASA Langley Research Center

AFFIRM Year 2 Focus

During year 2, our focus has been on experiments turning (fault-tolerant distributed) system specifications in an architectural DSL into transition system models.

Recall that the models we want to synthesize are:

- Efficient (small number of state variables)
- Representative of either synchronous or partially-synchronous systems
- Verifiable in a *mostly automatic* fashion
- Composable for use in modeling multi-level systems

Case Studies

During year 2 we focused mostly on two case studies:

- **OM(1)** - the classic **o**ral **m**essages with one round algorithm for byzantine fault tolerant validity and agreement,
- **WBS** - a real-time model of a **w**heel **b**reak **s**ystem that exhibits problematic sampling.

The two case studies entail quite different models. The OM(1) model explores message passing and data flow, while the WBS model explores the phase offset and period timing in multiple nodes observing a common signal.

A Calendar Based OM(1)

- Current state
- Fault model
- Verification
- Proof by induction
- Benchmarking

Calendar Based OM(1)

Much of our effort this year was targeted at finishing and refining all aspects of our OM(1) model in order to learn what is required in order to synthesize similar models.

In its current *finished* form, the model has these features:

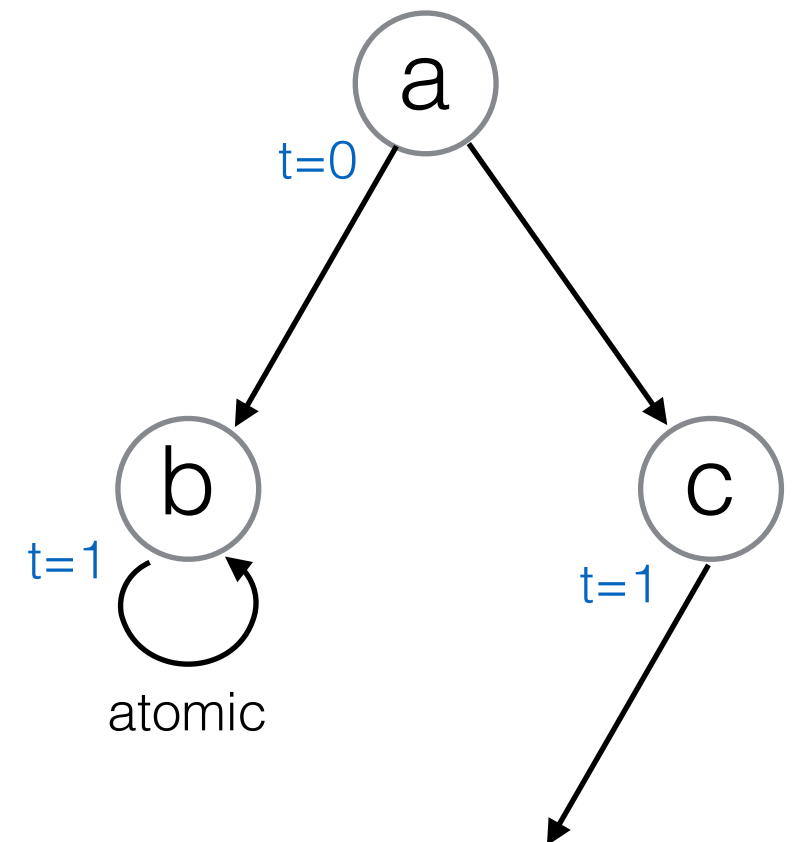
- quasi-synchronous approximation of the classical synchronous OM(1)
- the number of relays and receivers are parametrized

Calendar Based OM(1)

- message passing and time management is performed by a calendar automata that has one channel per (single duplex) communication channel in the specification
 - a notion of "atomic" transition block in the calendar automata framework greatly reduces exploration of interleaved transition steps
 - a full hybrid fault model is implemented along with the calendar framework, decoupled from any node behavior
- ➔ contains an inductive proof of key properties


Atomic Transition Blocks

- an atomic transition block is a sequence of coordinating transitions that occur **instantaneously**
- we can model local, stateful computations this way and avoid interleaving with network communication and remote computation



Fault Model

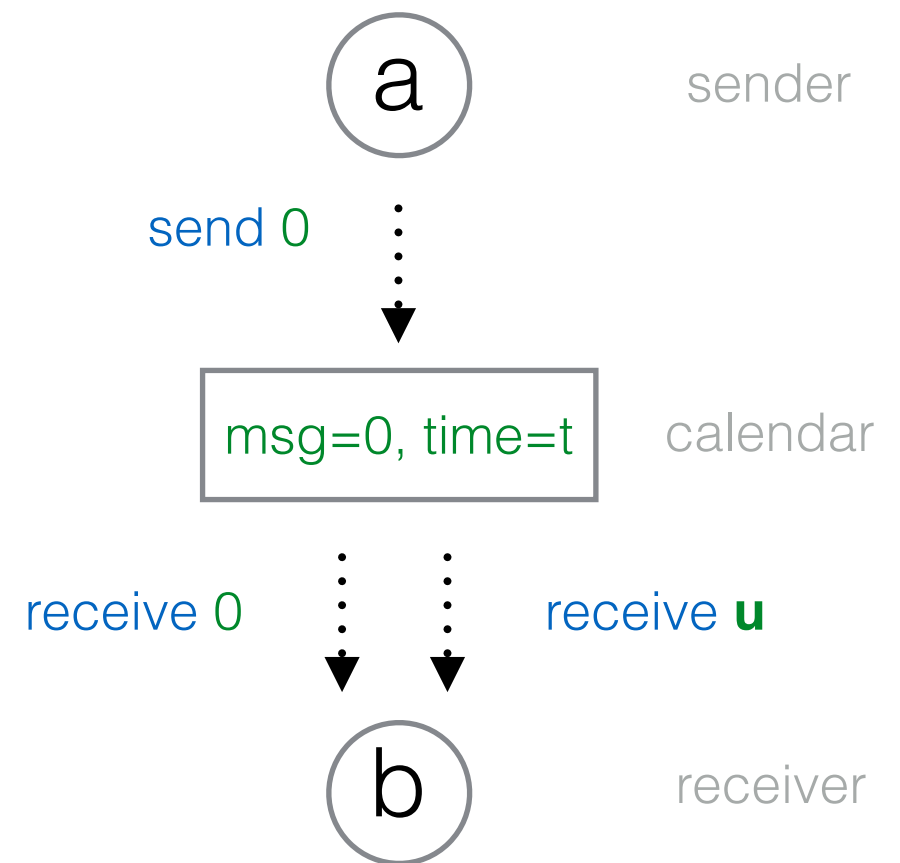
- We have implemented a full *hybrid fault model* on top of the calendar automata
- Fault types: byzantine, symmetric, manifest, none
- Messages are modeled by non-negative integers
- Each node is non-deterministically initialized faulty or not



-1	missing
0	good
>0	faulty

Fault Model

- When a node sends a message, it appear on the calendar as normal
- When a message is read, the result is either:
 - (sender is not faulty) the intended message
 - (sender is faulty) **uninterpreted constant** of type Message
- A-priori, the number of distinct messages generated in a trace is **unbounded**



u is a non-negative integer that we know nothing about
u is a (potentially) different constant on each receive

Fault Model

- Faults are manifested through reading from the global calendar
- Nodes in the model can be specified without any awareness of the fault model and don't need to be changed if the fault model changes
- The extent and type of faults is easily controlled from a single place in the model
- ... but use of uninterpreted constants limits our choice of model checking tools

Hybrid Fault Model

- The hybrid fault model accounts for four different fault behaviors: byzantine, symmetric, manifest, non-faulty
- The maximum fault assumption is an inequality between the weighted sum of faulty nodes and the total number of nodes

n=3	n=4
1 byzantine fault 1 symmetric fault up to 2 manifest faults	1 byzantine & 1 manifest fault or 1 symmetric & 1 manifest fault or up to 3 manifest faults

Faults at ADSL Level

```
test :: Tower p ()
test = Test "test1_period" $ do
  (in, out) <- channel
  per <- period (Microseconds 1000)
  m1 per in
  m2 out
  m3 out
```

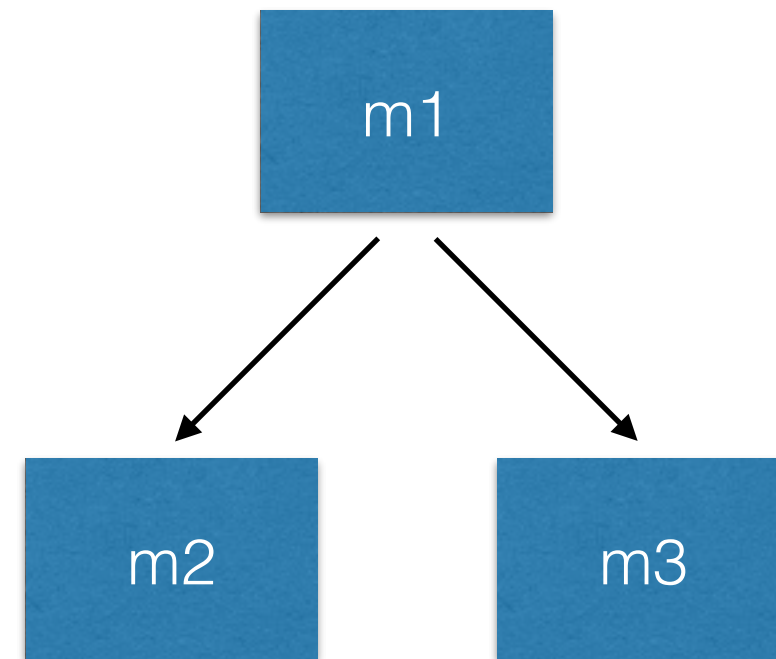
```
m1 per in = monitor "m1" $ do
  handler per "tick" $ do
    e <- emitter in 1
    callback $ \m -> do
      - - Some Ivory code
      emit e m
```

```
e <- byzEmitter in 1
```

```
m2 out = monitor "m2" $ do
  handler out "chan1msg" $
    callback $ \_ ->
      - - Some Ivory code
```

```
m3 out = monitor "m3" $ do
  handler out "chan1msg" $
    callback $ \_ ->
      - - Some Ivory code
```

every 1ms:
do work, put msg on channel



Verification

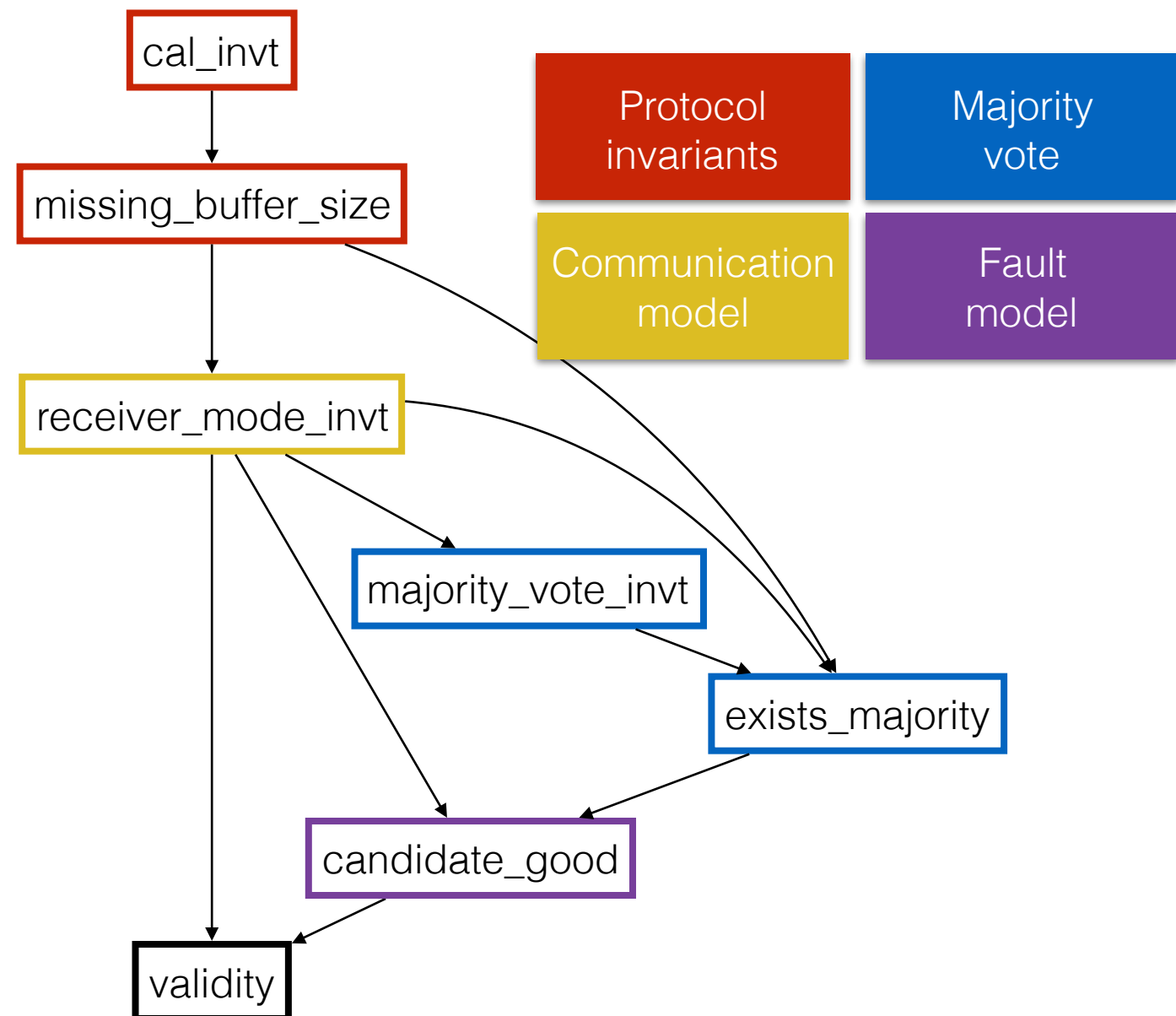
- For infinite models, we cannot use symbolic (BDD) model checking, nor ordinary bounded model checking
- *Infinite bounded model checking* is an SMT-based approach to model checking infinite systems
 - It is supported by SAL (`sal-inf-bmc`)
 - The main technique used is **induction**:
 - user provides inductive invariants of the system
 - model checker tries to prove a given property by induction using the invariants
 - ... or else provides a counter-example trace.
 - Proof by induction generally requires more work on the user's side (providing invariants)
 - Proof by induction is generally much faster and scalable

Proof by Induction

- We first completed a proof by induction of the **validity** property for OM(1)
 - Many auxiliary lemmas were required to complete the proof – they were hand generated
 - A main goal of the proof's construction was to separate out those auxiliary lemmas that we believe can be generated automatically in our translation
- As a side benefit, we have a proof which is much faster to verify and which scales much better than symbolic or ordinary bounded model checking

Lemma structure

- `cal_invt` : basic framework lemmas, e.g. *"time is always non-negative and increases monotonically"*
- `missing_buffer_size` : filled cells + unfilled cells equals the buffer size
- `receiver_mode_invt` : abstract sub-state machine for the receiver nodes
- `majority_vote_invt` : main invariant of the MJRTY (fast majority vote) algorithm
- `exists_majority` : invariant that holds when there is a majority in a receiver's buffer
- `candidate_good` : validity means voting a good value



Proof by Induction

We then completed a proof by induction of the **agreement** property for OM(1)

The proof involved two techniques not needed in the proof of validity:

- an abstract state machine (and associated invariant)
- history variables at points relevant to data flow

Abstract State Machine

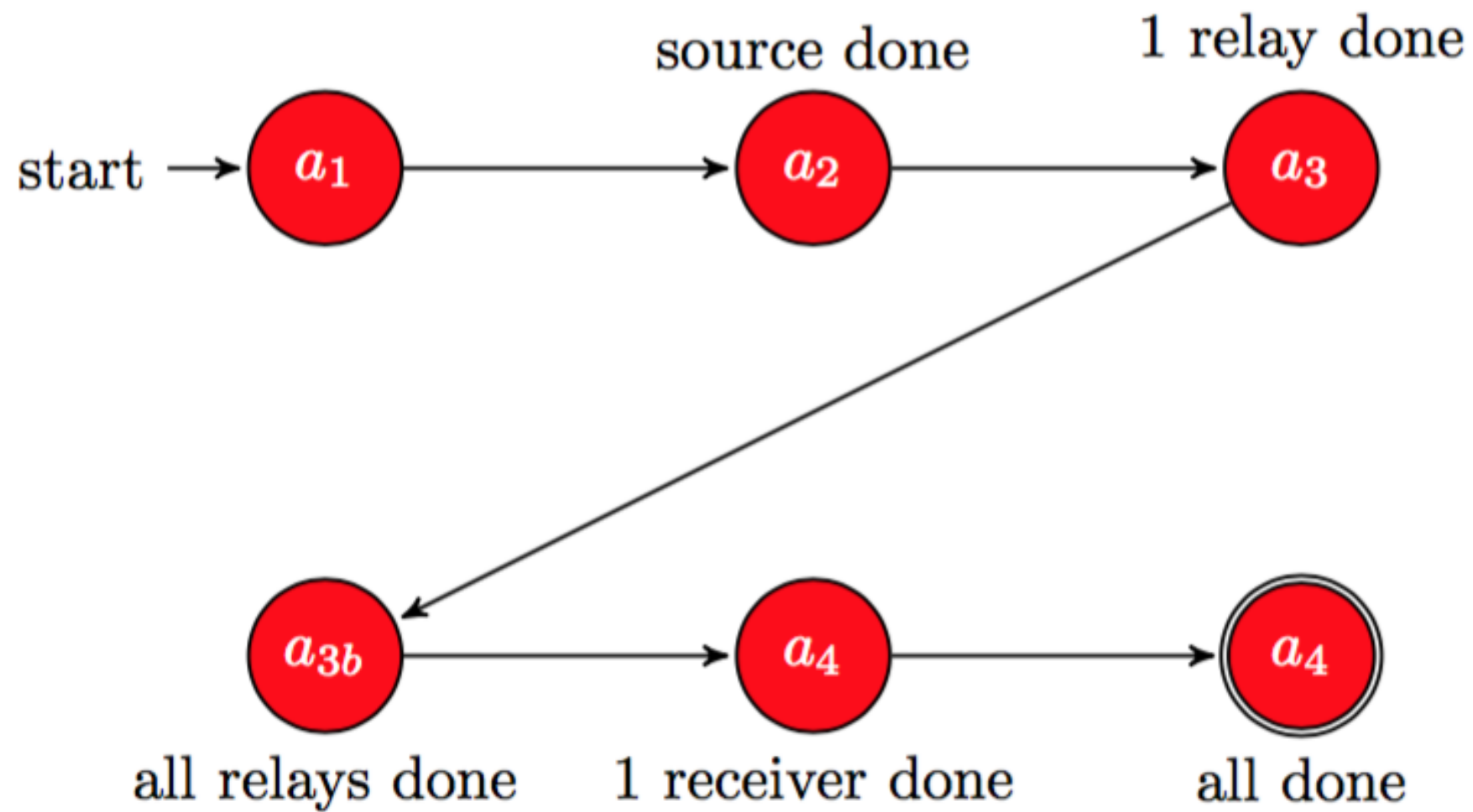
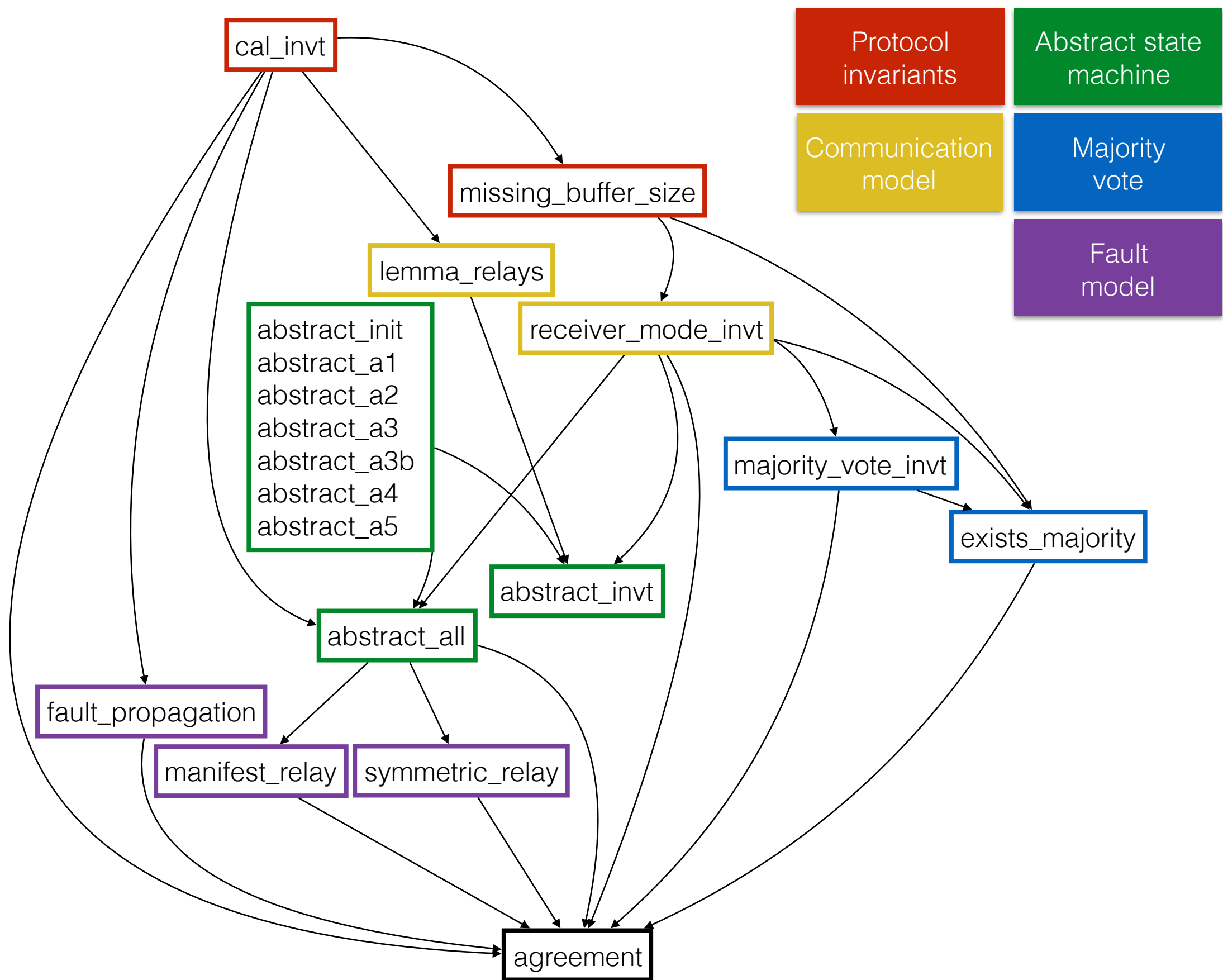


Figure 1



Benchmarking

- In order to compare our "synthesizable" model to the reference (Rushby's model) we benchmarked the verification over many sets of node configurations
- The largest reference model we were able to verify had 8 nodes
- The largest of our models we were able to verify had 10 nodes
- We see clear indications of single exponential time complexity verifying our model vs. double exponential for the reference

Benchmark

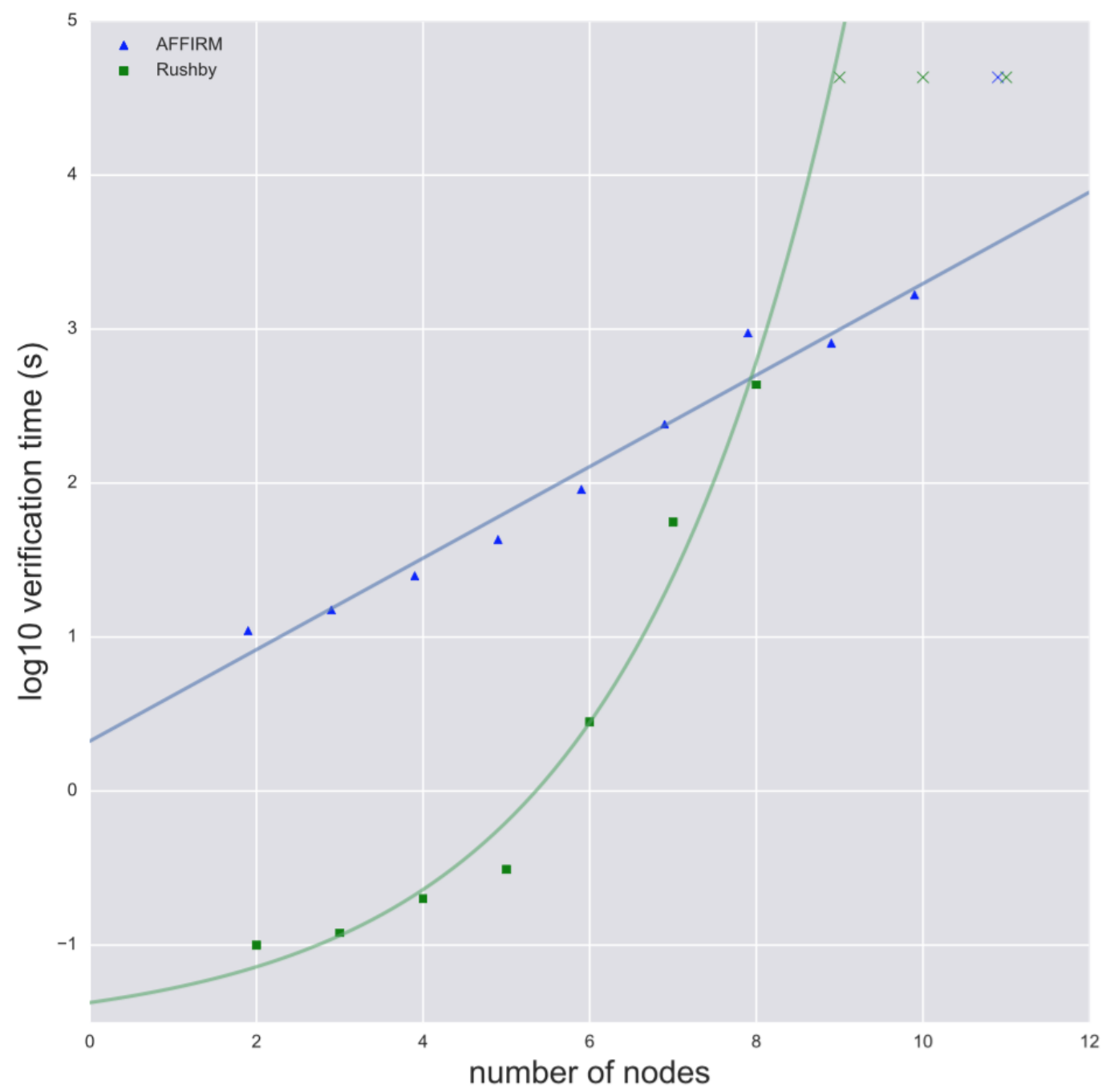


Figure 2

Beyond OM(1)

- Automatic lemma generation
- ADSL workbench prototype
- Year 3 plans

Automatic Lemma Generation

We have experimented with tools that provide lemma learning along with model checking

- Experiments with Rockwell Collins using JKind
- Reimplemented physical-layer protocol proofs (8N1, Biphase Mark) in Lustre
- JKind Implements *IC3 Modulo Theories via Implicit Predicate Abstraction* (<http://arxiv.org/pdf/1310.6847.pdf>), and k-induction
- Upshot: lemma generation only works for fixed constants, not uninterpreted constants

ADSL Workbench Prototype

We've been working towards a prototype with the following features:

- DSL with simple, intuitive syntax and a well-defined semantics
- C code generation backend
- SAL/Sally model generation backend
- Fault model configurable from the DSL
- Synchronous observers and abstract state machines that can be specified at a high level in the DSL

ADSL Workbench So Far

- As a starting point, we've chosen the Haskell DSL **Atom** ^[1]
- We added a typed, uni-directional communication channel as a primitive to Atom and augmented the C code backend accordingly
- We've produced an initial specification for OM(1) in Atom, patterned after our previous Ivory/Tower specifications

[1] Haskell EDSL for designing hard realtime embedded software
<https://github.com/tomahawkins/atom>

Source ("general") node in Atom

```
-- | Source node ("General")
source :: [V MsgType] -- ^ output channels
      -> Atom ()
source cs = period sourcePeriod
          . atom "source" $ do
    done <- bool "done" False
    let source_msg = goodMsg

    atom "source_poll" $ do
      cond $ not_ (value done)
      forM_ cs $ \c -> do
        c <== source_msg
      done <== Const True
```

Relay node in Atom

```
-- | Relay node ("Lieutenant")
relay :: Int          -- ^ relay id
      -> V MsgType    -- ^ input channel
      -> [V MsgType]  -- ^ output channel(s)
      -> Atom ()
relay ident inC outCs = period relayPeriod
                        . atom (tg "relay" ident) $ do
  done <- bool "done" False
  msg   <- msgVar (tg "relay_msg" ident)

  atom (tg "relay_poll" ident) $ do
    cond $ isMissing msg          -- we haven't stored a value yet
    cond $ not_ (isMissing inC)  -- there is a value available
    msg <== value inC
    forM_ outCs $ \c -> do
      c <== value inC
    done <== Const True
```

Year 3 Plans

- Finish work on an initial SAL/Sally backend for the DSL, including:
 - framework lemma generation
 - specification of properties
 - generation of observers and abstract state machines
- Specifying and modeling our other case studies in terms of the prototype workbench
- Investigate test generation for systems specified in the DSL
- Investigate modeling multi-level systems in the DSL
- Publishing the results of the project