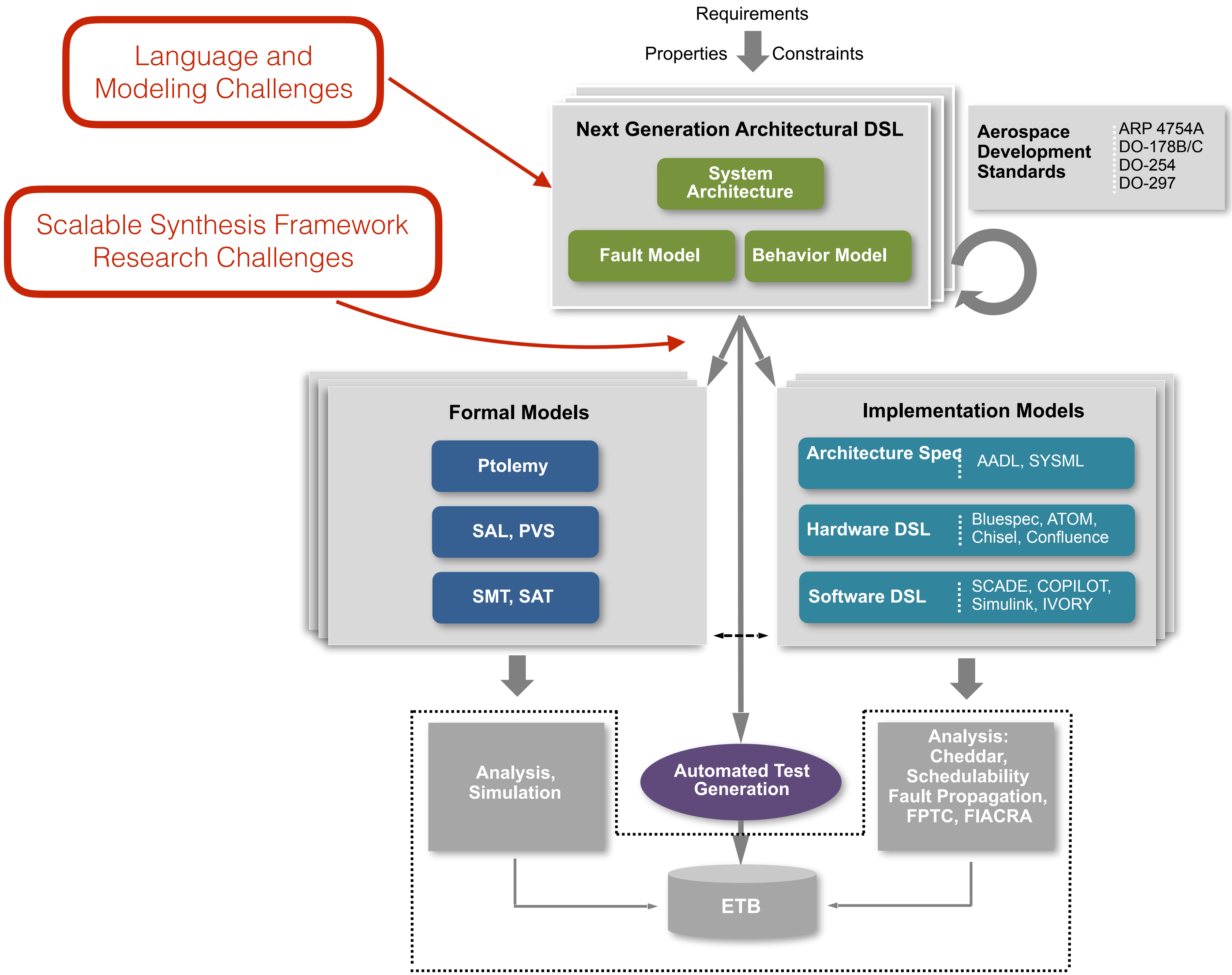
AFFIRM Year 2

August 9, 2016 - NASA Langley Research Center

Framework


Language and Modeling Challenges
Scalable Synthesis Framework Research Challenges
Requirements
Properties
Constraints
Next Generation Architectural DSL
System Architecture
Fault Model
Behavior Model
Aerospace Development Standards
ARP 4754A
DO-178B/C
DO-254
DO-297
Formal Models
Ptolemy
SAL, PVS
SMT, SAT
Implementation Models
Architecture Spec
AADL, SYSML
Hardware DSL
Bluespec, ATOM, Chisel, Confluence
Software DSL
SCADE, COPILOT, Simulink, IVORY
Analysis, Simulation
Automated Test Generation
Analysis: Cheddar, Schedulability Fault Propagation, FPTC, FIACRA
ETB

AFFIRM Year 1

- Determining what abstractions are appropriate and necessary in the ADSL
- Exploration of the case studies, primarily OM(1) and BRAIN
    - communication, timing, faults
    - not concerned (yet) with verification
- Researching related work, modeling languages and paradigms, etc.

AFFIRM Year 2

- Experiments toward translating system specifications in an architectural DSL into transition system models.

- Continuing to push on our case studies as they represent significant modeling and verification challenges.

- Added additional case study: WBS

- Converging on a suitable modeling framework (calendar automata, clock, and fault model)

What we've accomplished

- Preliminary ADSL implementation: AADL gen, C code gen
- 3 case-studies formalized (in different tools)
- Article on architecture + behavior (unpublished)
- Modeling abstractions (real-time, channel abstractions, faults) to support automated verification
- Full scalable, parameterized verification for channels + faults (OM(1) and WBS)

Case Studies

During year 2 we focused mostly on two case studies:

- **OM(1)** - the classic "oral messages with one round" algorithm for byzantine fault tolerant validity and agreement,

- **WBS** - a real-time model of a supposedly fault tolerant wheel break system that exhibits problematic sampling.

The two case studies entail very 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)

- Motivation
- Model highlights
- Deeper Dive
  - Fault model
  - Verification
  - Benchmarking

Motivation for studying OM(1)

- It is an algorithm/system with a long history; many models and much verification work has been done on it

  This means we can easily compare our models and results to a reference.

- Except for small instances, most of the existing verification work on OM(1) relies on interactive theorem proving.

- OM(1) is simple enough that if we can't scale verification here, it spells trouble for carrying out the larger project goals

Results

- First k-inductive model-checker verified proof for fully parametrized OM(1)

- Model / lemma decomposition

    - Node behavioral vs. messaging, clock, and sync framework

    - State machine for node behavior

    - Decoupled state machine for the message passing and clock framework

Highlights of our 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.

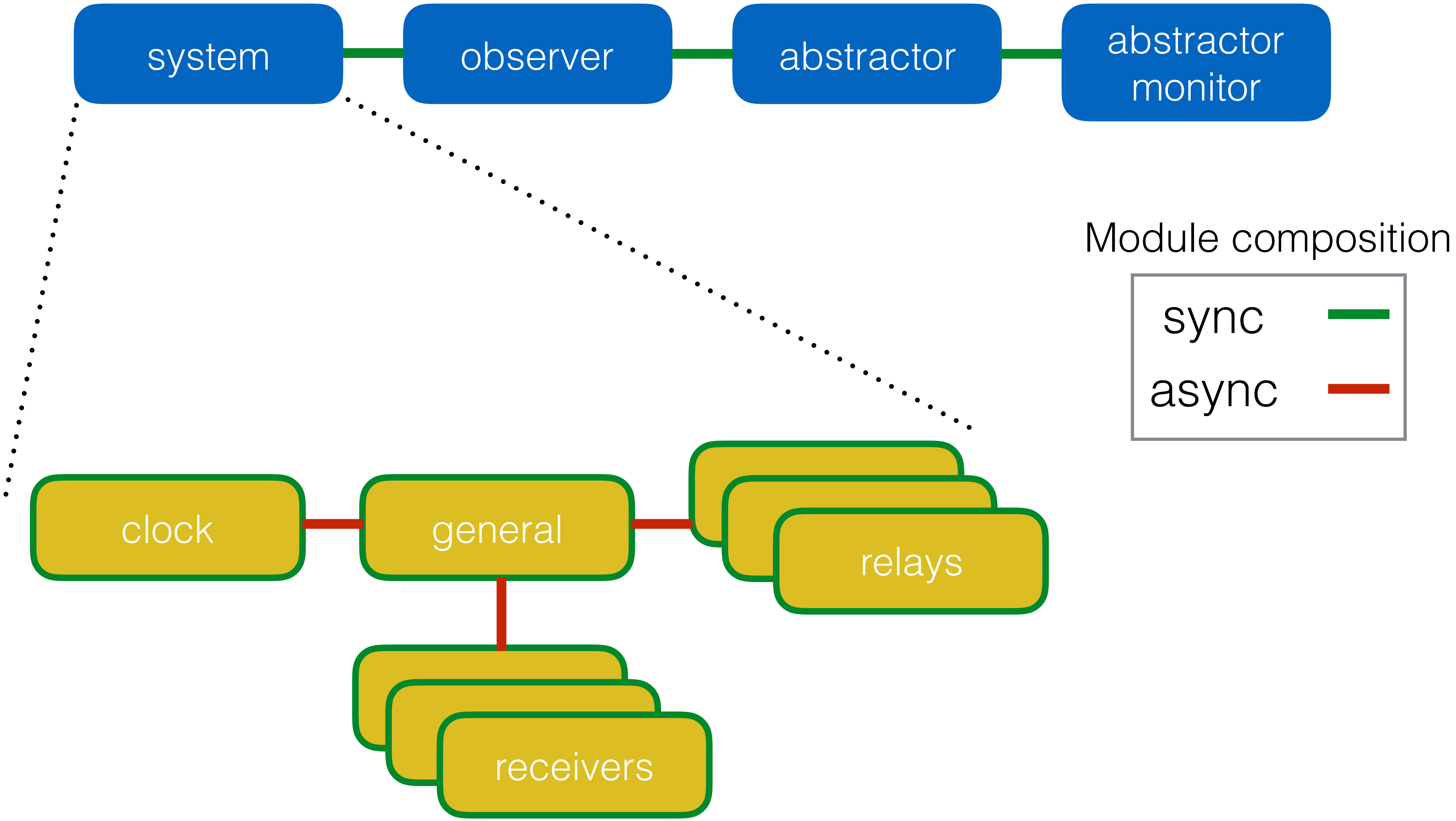
In its current form, the model has these features:

- Fundamentally asynchronous but we don't model any real time constraints
- Message sends and receives are separate (async) transitions
- Modeling synchronous OM(1) on a message passing fabric that is asynchronous
- Number of relays and receivers is parametrized (for the model and the proof)

Calendar Based OM(1)

- message passing and clock/time management is performed by a *calendar automata*
- a notion of "atomic" transition block that reduces exploration of interleaved transition steps
- a *hybrid fault model* is implemented along side the calendar, decoupled from node behavior
- a *k-inductive proof* of validity and agreement for each node configuration

Model Architecture



system

observer

abstractor

abstractor monitor

clock

general

relays

receivers

Module composition

sync	🟩
async	🟥

Fault Model

- We have implemented a hybrid fault model in our framework
- We developed a fault model with two principles in mind:
    - separation of node behavior and fault behavior
    - avoiding the need for ad-hoc reasoning

Avoiding ad-hoc reasoning

> " To achieve the full range of faulty behaviors, it seems that a faulty source should be able to send a different incorrect value to each relay, and this requires n different values. ...
>
> ... It might seem that we need some additional incorrect values so that faulty relays can exhibit their full range of behaviors. ... "

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

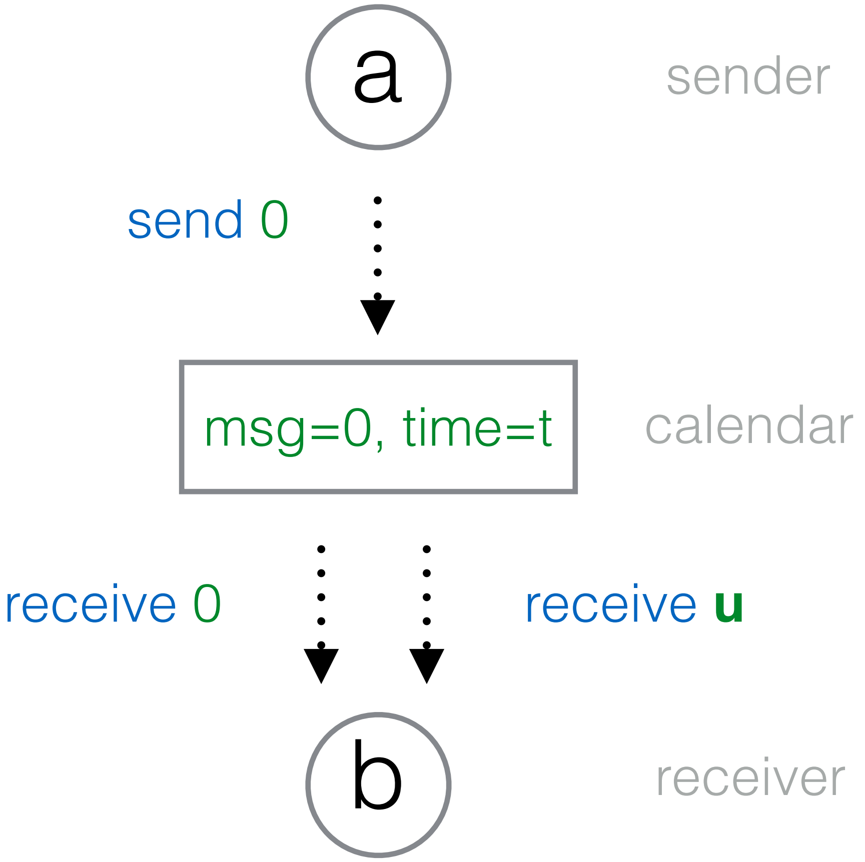
-1	missing
0	good
>0	faulty

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***

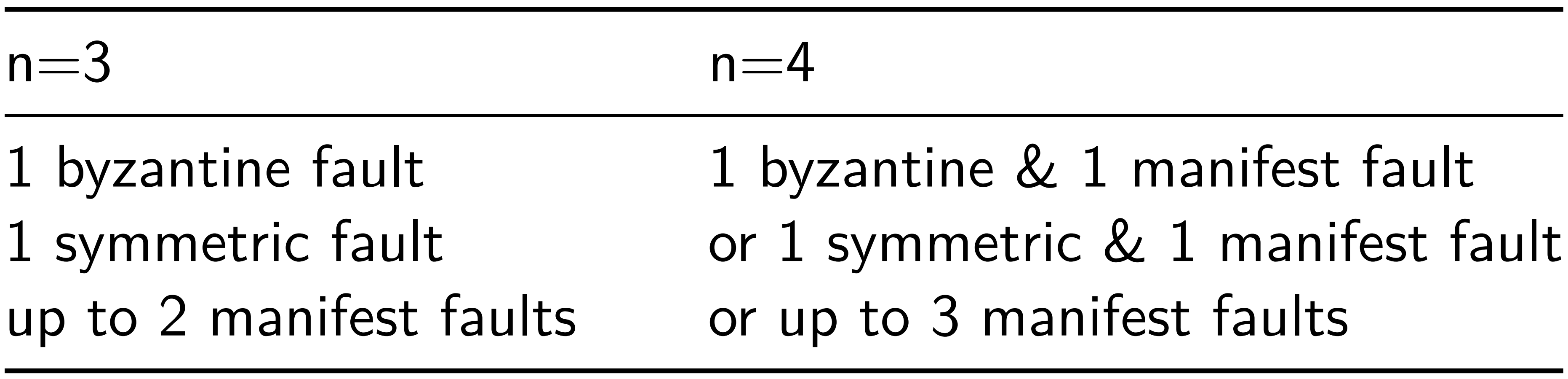


u is a non-negative integer that we know nothing about
u is a (potentially) different constant on each receive

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

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

Verification

Verification

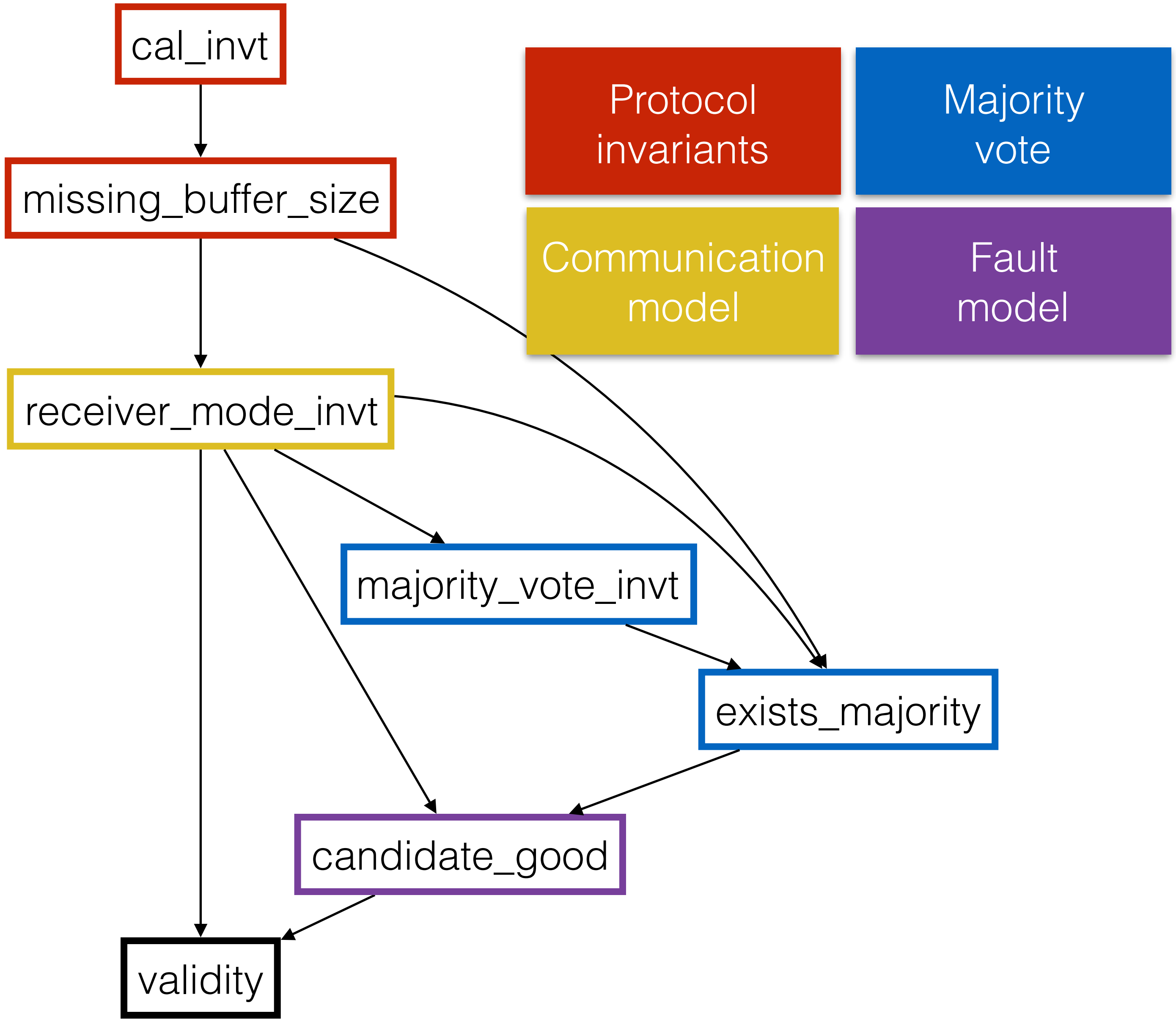
- For infinite models, we cannot use symbolic (BDD-based) model checking, nor ordinary bounded model checking
- *Infinite bounded model checking* is an SMT-based approach to model checking infinite systems
    - It is supported by SAL (`sal-inf-bmc`)
    - The main technique used is **induction:**
        - 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

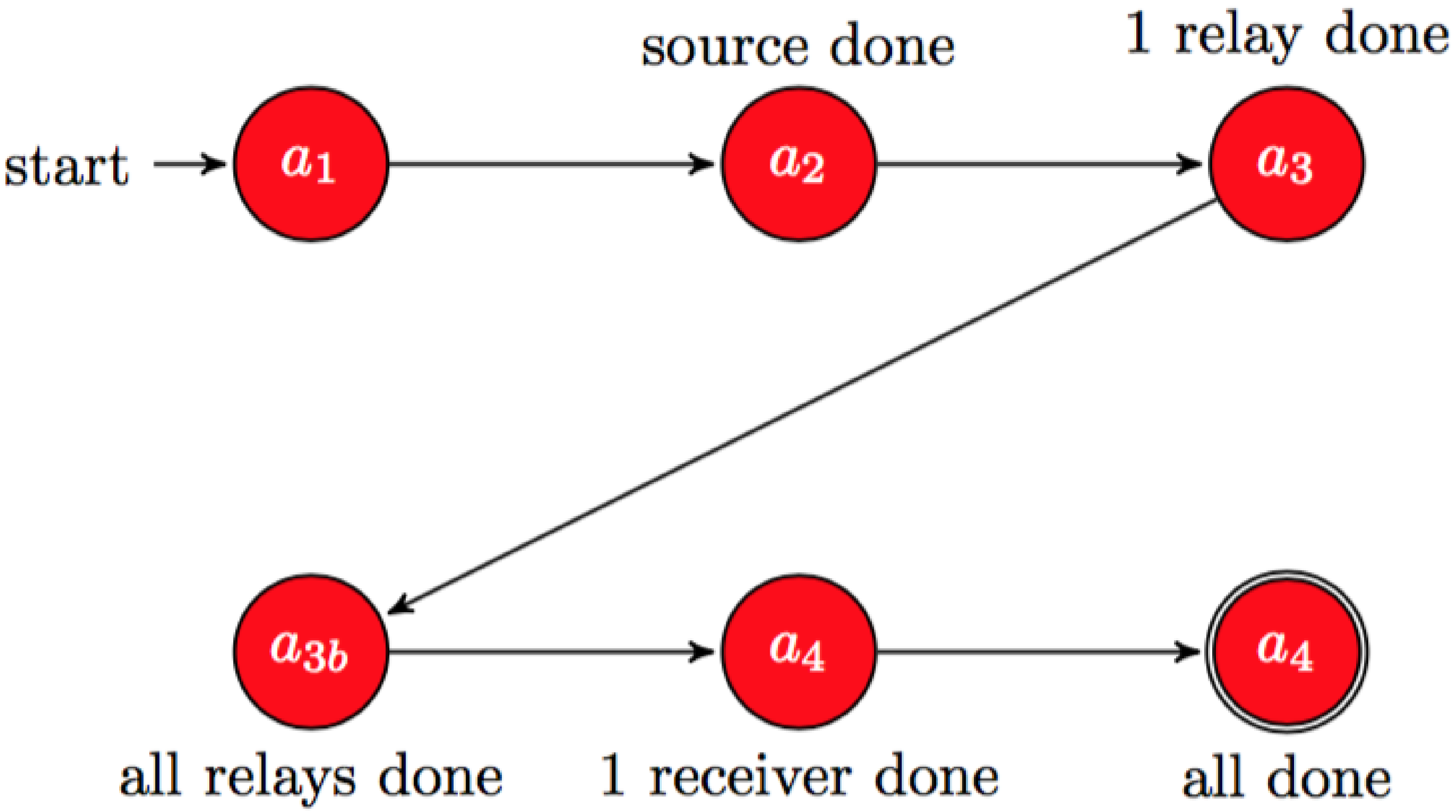
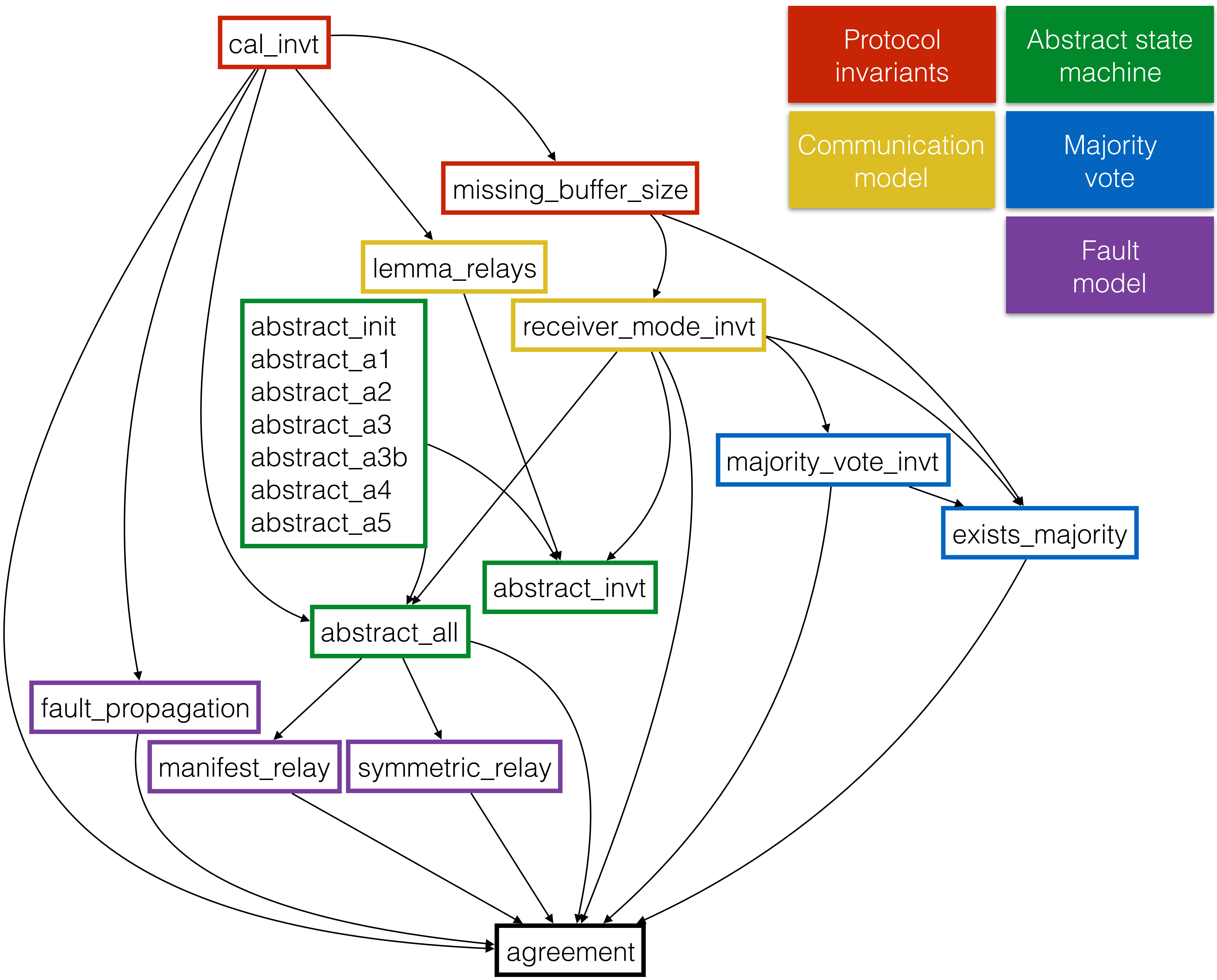


Figure 1



Protocol invariants

Abstract state machine

Communication model

Majority vote

Fault model

cal_invt

missing_buffer_size

lemma_relays

receiver_mode_invt

abstract_init
abstract_a1
abstract_a2
abstract_a3
abstract_a3b
abstract_a4
abstract_a5

majority_vote_invt

exists_majority

abstract_invt

abstract_all

fault_propagation

manifest_relay

symmetric_relay

agreement

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

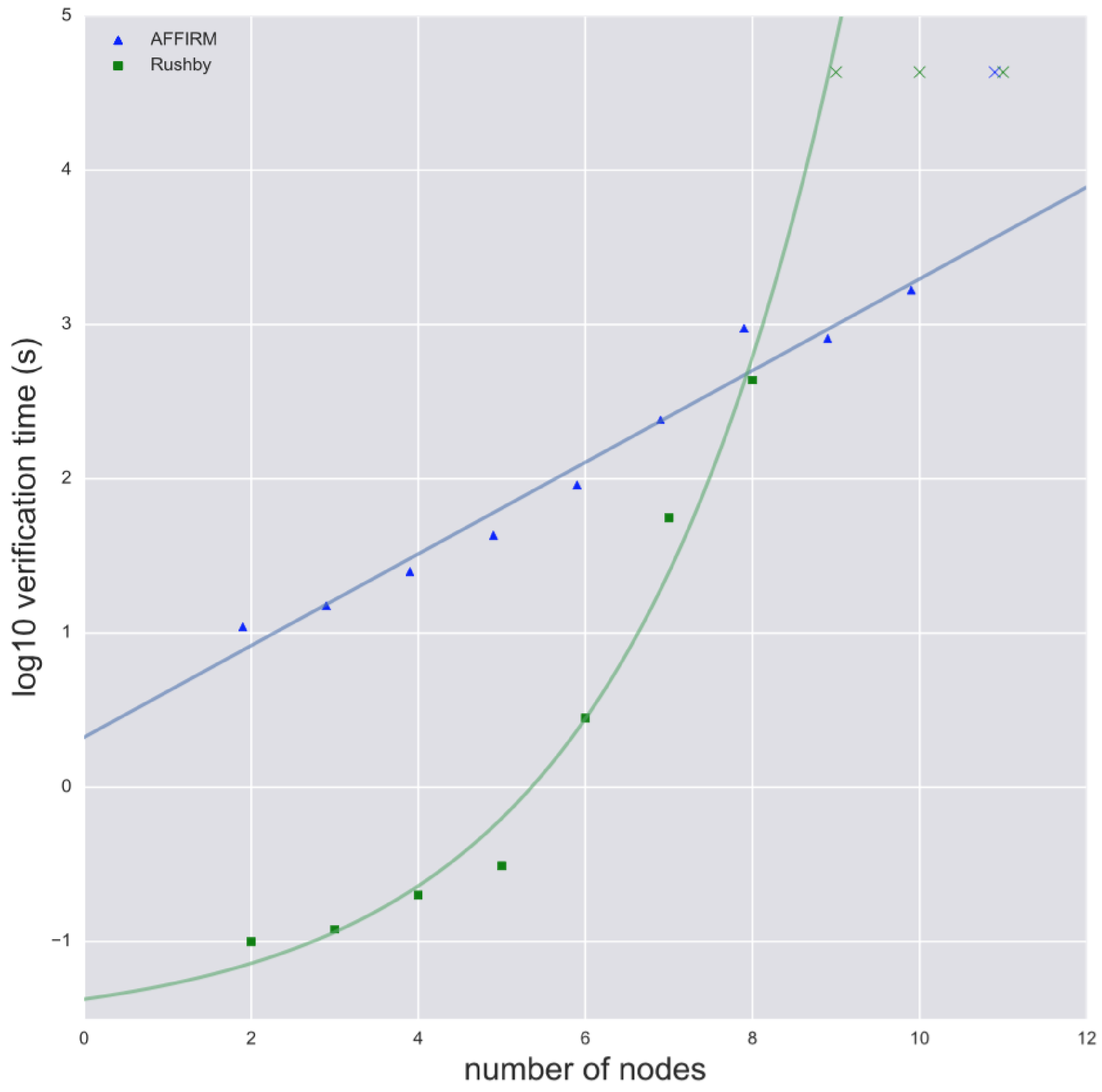


Figure 2

WBS Case Study…

Beyond Case Studies

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

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

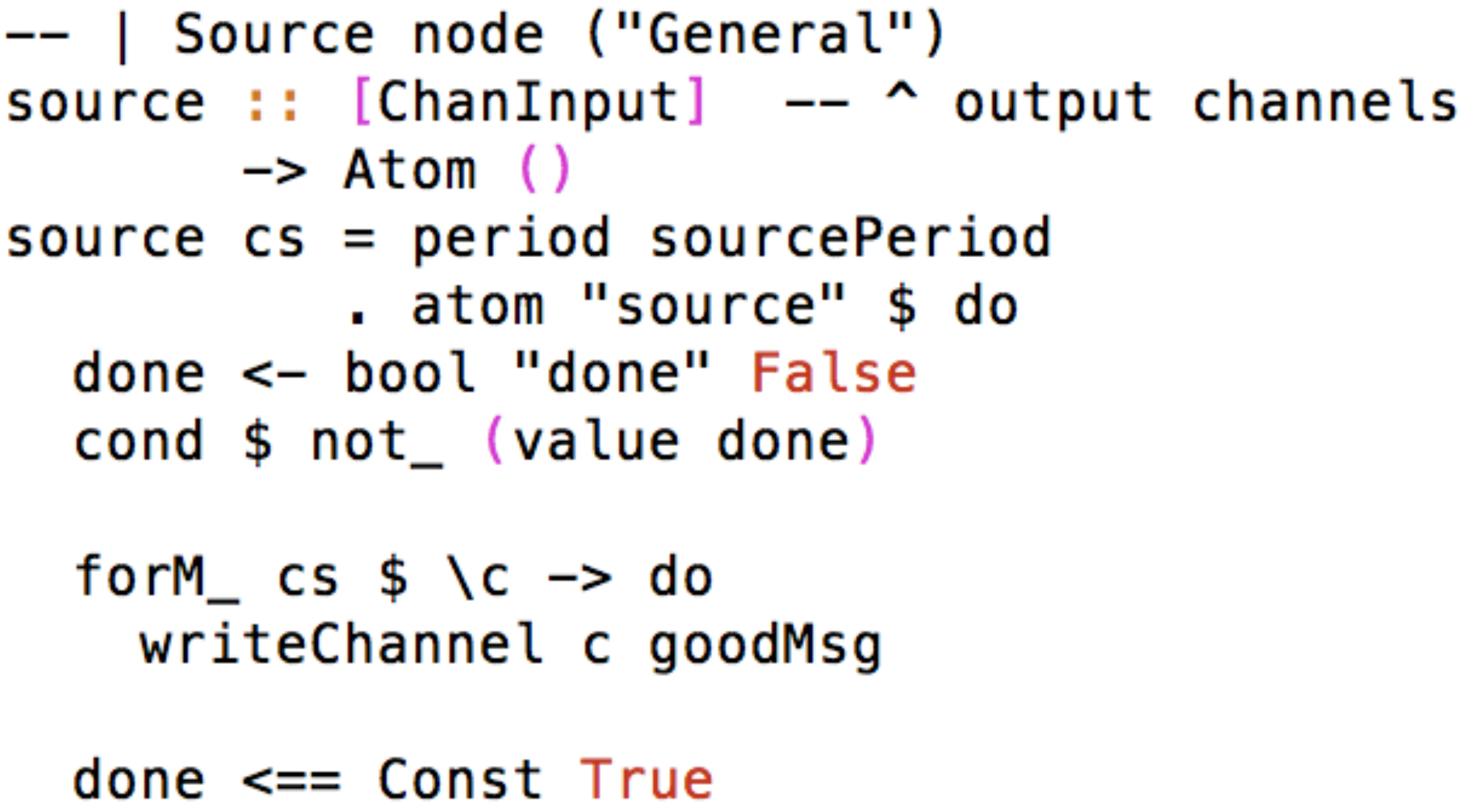
ADSL Workbench So Far

- We've simplified the DSL syntactically and lessened the scope of the expression language.

- We added typed, uni-directional communication channels as a primitive whose semantics match our message passing model.

- We've produced refined specification for OM(1) in the ADSL.

Source node in the ADSL

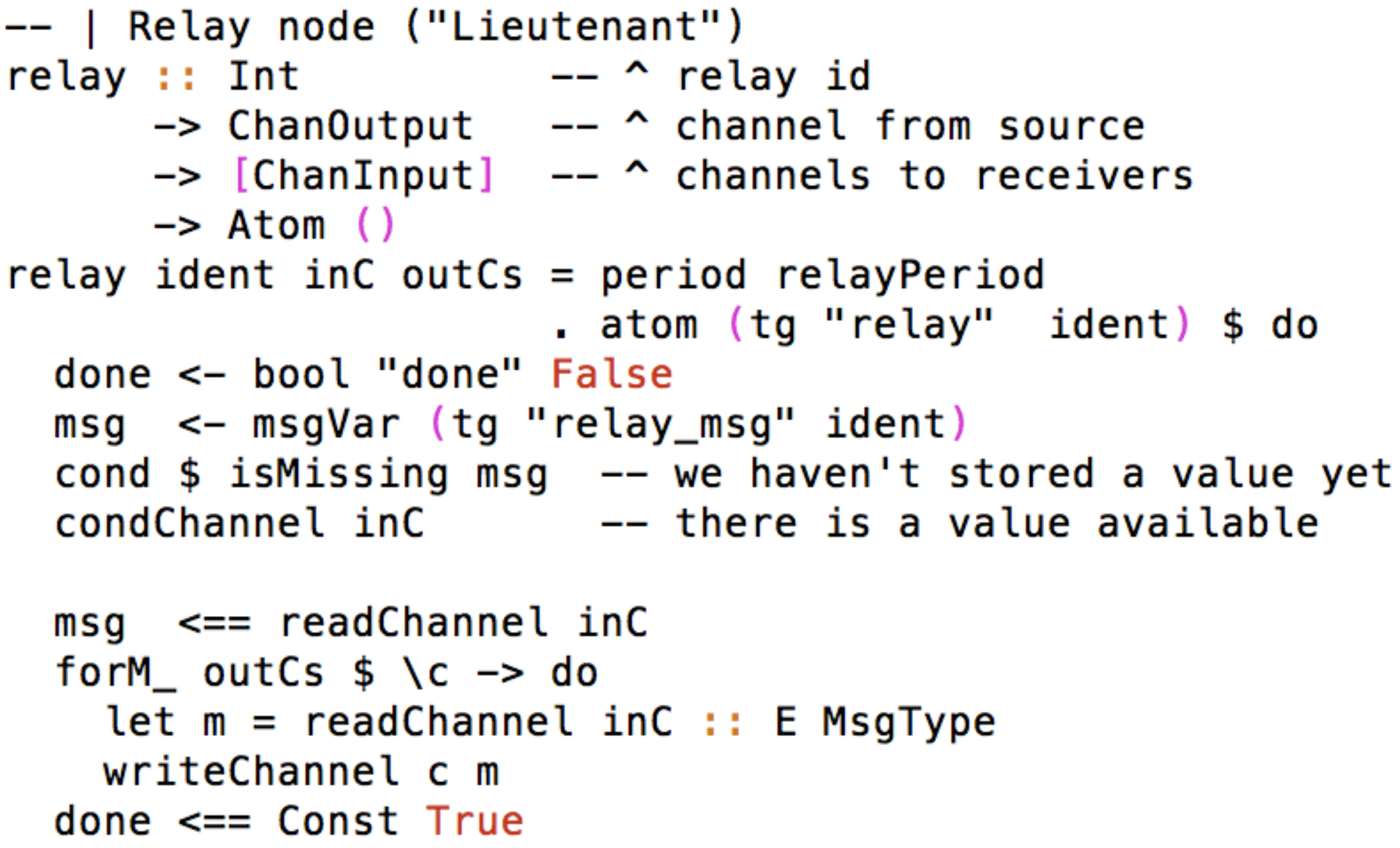


```
-- | Source node ("General")
source :: [ChanInput]  -- ^ output channels
       -> Atom ()
source cs = period sourcePeriod
          . atom "source" $ do
  done <- bool "done" False
  cond $ not_ (value done)

  forM_ cs $ \c -> do
    writeChannel c goodMsg

  done <== Const True
```

Relay node in Atom



```
-- | Relay node ("Lieutenant")
relay :: Int            -- ^ relay id
      -> ChanOutput     -- ^ channel from source
      -> [ChanInput]    -- ^ channels to receivers
      -> Atom ()
relay ident inC outCs = period relayPeriod
                      . atom (tg "relay"  ident) $ do
  done <- bool "done" False
  msg  <- msgVar (tg "relay_msg" ident)
  cond $ isMissing msg  -- we haven't stored a value yet
  condChannel inC       -- there is a value available

  msg  <== readChannel inC
  forM_ outCs $ \c -> do
    let m = readChannel inC :: E MsgType
    writeChannel c m
  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
- Adding case study of TTE as a multi-level system
- Investigate <u>test generation</u> for systems specified in the DSL
- Investigate modeling <u>multi-level systems</u> in the DSL
- Publishing the results of the project