Modular Model-Checking of a Byzantine Fault-Tolerant Protocol [1]

Benjamin F Jones[2] and Lee Pike[2]

[1]Supported by NASA Contract NNL14AA08

[2]Galois, Inc. – http://galois.com

Challenge

Model checking distributed, fault-tolerant systems is challenging:

- large state spaces
- non-determinism
- complexity inherent in modeling interacting nodes subject to various fault modes

Coping with complexity

Model-checking these systems has been accomplished by resorting to ad-hoc abstractions:

- local node behavior is abstracted (e.g. replaced by pre/post conditions)
- message passing is simplified using shared state
- systems are modeled at small, fixed scales

Ad-hoc abstraction tends to cause a model to *diverge from an implementation*, reducing its applicability.

Modularity

- Model-checking these systems is further complicated when system behavior and fault behavior are mixed in the model.
- *Compositional reasoning*: reason about node behavior and fault model separately, then compose the proofs

What We Want in a Model

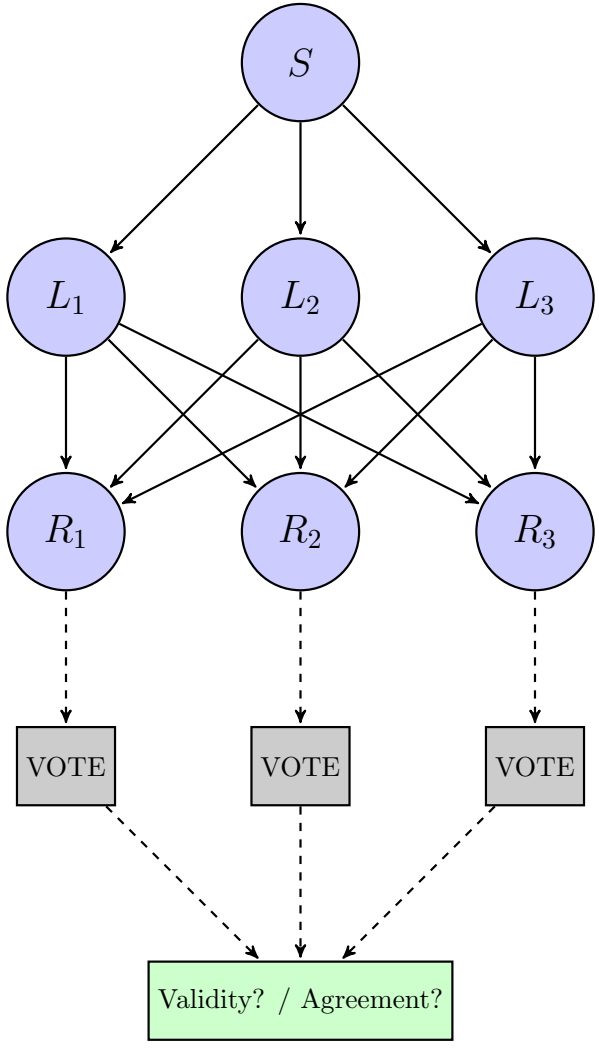
The premise for this work is that we want, *ideally*, to produce formal models of distributed systems that:

1. have implementation-level detail,
2. are tractable to the model checker,
3. and are modular to the extent possible.

What We've Done

- **Framework**: We've attempted to come up with a fairly general framework for modeling distributed, fault-tolerant systems in a way that gets us closer to the ideal of the 3 last points.
- **Case Study**: We've taken this framework and used it to model variations on the *Oral Messages* algorithm that vary in their timing model, node behavior, and fault model.
- First model-checked verification of $\mathrm{OMH}(1)$ that:
  - does not elide implementation level detail
  - is parameterized over the number of nodes (verified up to 12 nodes)
  - has a modular, inductive proof

Oral Messages with 1 Round



Hybrid Fault Model

Thambidurai and Park introduced their "Hybrid Fault Model" [3] which captures 4 different fault modes and a system for weighting their impact.

- Non-faulty
- Manifestly faulty
- Symmetrically faulty
- Byzantine faulty

$\mathrm{OM}(1)$ was shown to be tolerant of a certain weighted sum of these fault types. In the classical case where only byzantine faults are considered and their are 3 lieutenants, the system tolerates at most 1 fault.

---

[3] P. Thambidurai and Y.K. Park. *Interactive consistency with multiple failure modes*. Symposium on Reliable Distributed Systems, 1988.

Hybrid Oral Messages Algorithm

- In the course of formalizing this result on fault-tolerance, it was discovered [4] that the proof was wrong. However, the $\mathrm{OM}(1)$ algorithm can be adapted to one that satisfies the conclusion of Thambidurai and Park.
- This algorithm, due to Lincoln and Rushby, is called $\mathrm{OMH}(1)$.

---

[4] P. Lincoln and J. Rushby. *A formally verified algorithm for interactive consistency under a hybrid fault model*. In Fault Tolerant Computing Symposium 23, 1993.

Three Systematic Abstractions

In this framework for modeling systems, we use three principal abstractions:

1. Calendar automata
   - modeling message passing, including details like real time, message delay, buffering, ...
2. Symbolic fault injection
   - injecting faults into the model in a way that is decoupled from local node behavior
3. Abstract transition systems
   - strengthening invariants needed to verify properties of the system without having to abstract actual system behavior

Symbolic Fault Injection

- typical approach to modeling faults: add a new state variable to for each node, indicating fault state. The node's logic then acts on this value.

State

- $a, b, \ldots$

Fault State

- f

Action

if f then faulty_behavior()
else normal_behavior()

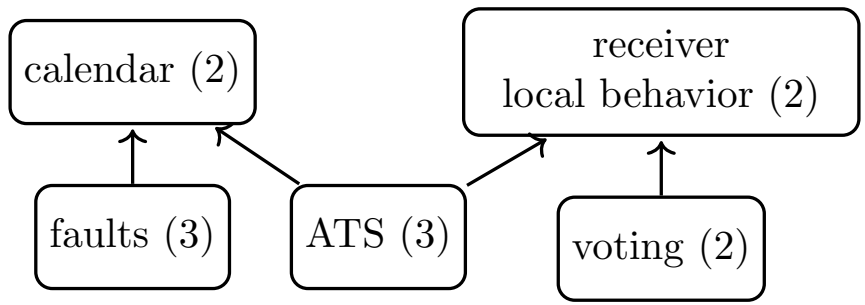
Fault Injection

Fault injection divorces fault behavior from node behavior by injecting faults at the channel level.

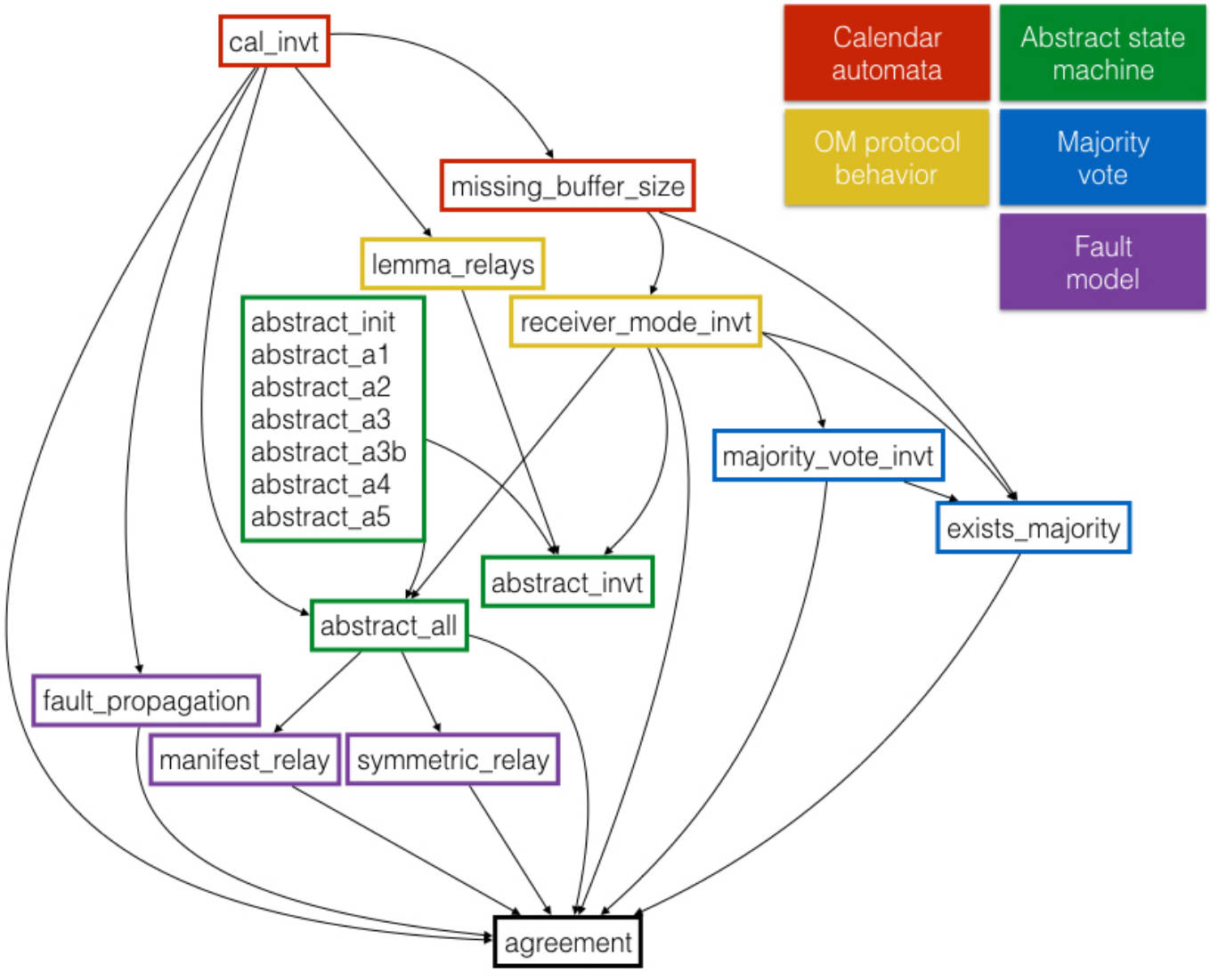
- faulty values are represented as uninterpreted constants
- faulty values are constrained depending on the fault type
- separation of concerns allows us to reason about the fault model in isolation from local node behavior

Verification: Lemma Diagram

Our verification of the case study proceeds by k-induction, using SAL [5]. We strengthen the inductive hypothesis using hand-crafted lemmas which are organized as follows:



[5] SAL. Computer Science Lab, SRI International. `http://sal.csl.sri.com`


Calendar automata
Abstract state machine
OM protocol behavior
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

Modularity: Variant Models

To assess how modular our modeling framework is, we experimented on our case study, adding or changing features of the model and measuring how much effort was required to make the changes and verify the result.

1. Omissive assymetric faults
2. Time triggered message passing
3. Mid-value selection voting

Omissive Assymetric Faults

Consider changing the fault model:

- Removing all faults from the model is easy: we simply set the maximum weighted sum in the maximum fault assumption to zero.
- Adding a new fault mode is more work, but is still modular:
- we add a new uninterpreted function definition for omissive asymmetric faults
- we modify the fault type definition in SAL
- finally, we define the effect of this new fault type on the calendar

	Definitions (58 total)	Invariants (11 total)	Invariant classes (5 total)
Omissive Asymmetric Faults	1 new, 2 modified	2 modified	faults

Future Work

As part of NASA Contract NNL14AA08 we are working on a modeling workbench (DSL and surrounding tools) to support modeling in the framework described here.