Modular Model Checking of a Distributed, Fault-tolerant System

Challenge

Model checking distributed, fault-tolerant systems is challenging:

- large state spaces
- non-determinism
- complexity inherent in modeling multiple nodes behavior and faults

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 abstractions tend to cause models to *diverge* from implementations, reducing their applicability.

Modularity

TODO rewrite
Model checking these systems is further complicated when system behavior and fault behavior are mixed.
Similarly, when trying to prove that a model satisfies a property, it is very useful to be able to reason about the system behavior and the fault behavior independently and then compose the proofs. The same goes for local node behavior versus aspects of the communication model.

What We Want in a Model

1. Implementation-level detail
2. Scalability of model checking
3. Modularity

What We've Done

- 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
- We've taken this framework and used it to model variations on the *Hybrid Oral Messages* algorithm that vary in their timing model, node behavior, and fault model.

Oral Messages Algorithm

Oral Messages with 1 Round

TODO diagram

Hybrid Fault Model

In [1], T and Park introduced their "Hybrid Fault Model" which captures 4 different types of node fault and a system for weighting their impact.

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

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.

Hybrid Oral Messages Algorithm

In the course of formalizing this result on fault-tolerance, it was discovered [2] that the proof was wrong. However, the OM(1) algorithm can be adapted to one that satisfies the conclusion of [1].
This algorithm, due to Lincoln and Rushby, is called OMH(1).

Three Systematic Abstractions

In this framework for model 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: new state vars for fault state, node acts upon this value
- fault injection:
- divorce fault behavior from node behavior by injecting faults at the channel level
- faulty values are represented as uninterpreted constants
- faulty values are constrained as a function of the fault type
- separation of concerns allows us to reason about the fault model in isolation from local node behavior

Four Versions of OMH(1)

Deep Dive on One Such Version

Lemma Diagram

Lemma Diagram for Variant

References

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

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