Modular Model-Checking of a Byzantine Fault-Tolerant Protocol

Benjamin F Jones and Lee Pike

Galois, Inc., Portland OR 97204
{bjones,leepike}@galois.com

Abstract. With proof techniques like IC3 and k-induction, model-checking scales further than ever before. Still, fault-tolerant distributed systems are particularly challenging to model-check given their large state spaces and non-determinism. The typical approach to controlling complexity is to construct ad-hoc abstractions of faults, message-passing, and behaviors. However, these abstractions come at the price of divorcing the model from its implementation and making refactoring difficult. In this work, we present a model for fault-tolerant distributed system verification that combines ideas from the literature including calendar automata, symbolic fault injection, and abstract transition systems, and then use it to model-check various implementations of the Hybrid Oral Messages algorithm that differ in the fault model, timing model, and local node behavior. We show that despite being implementation-level models, the verifications are scalable and modular, insofar as isolated changes to an implementation require isolated changes to the model and proofs. This work is carried out in the SAL model-checker.

1 Introduction

Fault-tolerant distributed systems are famously complex, yet are the backbone of life-critical systems, such as commercial avionics. Consequently, this class of systems demands high-assurance of correct design and implementation. Formal verification can help provide that assurance.

The verification of this class of systems has usually been at the algorithmic level, eliding details about a concrete implementation. Historically, it has relied on formal models verified by interactive theorem-proving [1, 2, 3, 4]. If formal verification is to be introduced into the workflow of system designers, though, we need more automated methods that scale for implementation-level models. (Mostly) automated proof techniques are required to reduce the need for specialized verification expertise. We also need programmatic verification of *implementations*. System designers create software and hardware implementations to test, simulate, and deploy. Discrepancies between implementations and algorithmic models can arise if the latter is abstracted too much from the former [5], particularly if those abstractions are ad-hoc and system specific. Furthermore, as implementations are modified to explore the design space, it is easy for the

formal model and the implementation to become inconsistent, so the verification is no longer about the system deployed.

There are at least two classes of abstractions that separate protocol-level models of fault-tolerant distributed algorithms from their implementations. One is to intertwine the environmental model with the system description. For example, the behaviors of nodes are naturally specified as a transition system in which transitions are guarded by the node's fault state. But faults are part of the environment; an implementation does not typically use its own fault status to choose actions! Another class of abstractions is used to simplify models. For example, message passing might be abstracted with shared state, or a node's local behavior is elided and instead, the output is constrained by a *specification* of the behavior.

In this paper, we present a fault-tolerant distributed systems model, and use that model to verify several variant implementations of the Byzantine fault-tolerant Hybrid Oral Messages algorithm (OMH) [3]. The model combines various ideas from the literature to build scalable and modular formal models suitable for infinite-state model-checking, and it reduces the need for ad-hoc abstractions and optimizations. In Section 2, we present the important aspects of the model, including calendar automata, originally developed by Dutertre and Sorea [6], symbolic fault-injection, and abstract transition systems for verification.

We use the model to verify implementation-level models of OMH in which message passing is explicit, nodes are not forced to execute strictly synchronously, and voting is explicit. In short, the models corresponds closely with an implementation of the algorithm. In Section 3, we first describe OMH, then an implementation of it that uses the Boyer-Moore Fast Majority Vote algorithm (Fast MJRT) [7]. We then describe a set of modular invariants, such that the invariants only concern specific aspects of the model (e.g., faults, local node behavior, or the passage of time). The verification is interesting in its own right, as it is the first fully parametric (on the number of nodes) model-checked *implementation* of the algorithm.

In Section 4, we first show that despite being implementation-level, the model is scalable. Developing invariants requires some user guidance, and isolated changes to an implementation should require isolated modifications to the model and proof. To demonstrate this, we modify the OMH implementation along the dimensions of faults (by adding an omissive-asymetric fault type [8]), time (by making a time-triggered model), and local behavior (by changing the majority vote to a mid-value selection) and show that in each case, the modifications are small and modular.

Our primary contributions are (1) a model-checking verification of an OMH implementation, and (2) demonstrating that our modeling paradigm allows for modular verification. Additionally, the idea of symbolic fault injection (Section 2.2) is novel.

Finally, in Section 5 we describe related work, and we make concluding remarks in Section 6.

The models and experiments reported herein can be found online.[1]

2 Formal Model

Here we describe our formal model specialized for fault-tolerant distributed systems. The model draws on three principal abstractions: calendar automata, symbolic fault injection, and abstract transition systems; we describe each below.

2.1 Calendar Automata

Real-time system verification in general-purpose model-checkers requires an explicit formalism of real-time progression. Trying to encode real-time clocks directly is difficult; in particular, one must avoid Zeno's paradox in which no progress is made because state transitions simply update real-valued variables by an infinite sequence of decreasing amounts whose sum is finite. To avoid this problem, Dutetre and Sorea developed *calendar automata* [6], which is itself inspired by event calendars used in discrete-event simulation. Rather than encoding "how much time has passed since the last event", it encodes "how far into the future is the next scheduled event", and a real-valued variable representing the current time is updated to the next event time.

Define a set of *events* $e_0, e_1, \ldots, e_n \in E$. For now, we do not define events; intuitively, an event is a set of state variables (shortly, we will associate events with messages sent in a distributed system). When an event is *enabled*, the transitions over events are enabled; otherwise, the variables stutter (maintain the same value).

An *event calendar* $\{(e_0, t_0), (e_1, t_1), \ldots, (e_n, t_n)\}$ is a set of ordered pairs (e_i, t_i) called *calendar events* where $e_i \in E$ is an event and $t_i \in \mathbb{R}$ is a *timeout*, the time at which the event is scheduled. We denote element (e_i, t_i) of an event calendar by c_i.

Let *cal* be an event calendar and $c_i, c_j \in cal$ be calendar events. Define an ordering on calendar events such that $c_i \leq c_j$ iff $t_i \leq t_j$, and $\min(cal) = \{c_i | \forall c_j \in cal,\ c_i \leq c_j\}$ are the minimum elements of *cal*.

Let a transition system $\mathcal{M} = (S, I, \rightarrow)$, be a set of states S, a set of initial states $I \subseteq S$, and a transition relation $\rightarrow \subseteq S \times S$. We implicitly assume a set of state variables such that each state $\sigma \in S$ is a total function that maps state variables to values. We sometimes prime a state to denote that it satisfies the transition relation: $\sigma \rightarrow \sigma'$. We also sometimes use a variable assignment notation to describe what state variables are specifically updated: e.g., $\sigma' = \sigma[v := v + 1]$.

We distinguish two special state variables in a transition system: (1) $now \in \mathbb{R}$ denotes the current time in the state, and (2) *cal* is an event calendar.

The following laws must hold of a transition system $\mathcal{M}$ implementing a calendar automaton:

[1] https://github.com/GaloisInc/mmc-paper

1. Time is initialized to be less than or equal to every calendar timeout: $\forall \sigma \in I$, $\forall (e_i, t_i) \in \sigma(cal)$, $\sigma(now) \leq t_i$.
2. In all states, if the current time is strictly less than every calendar event, then the only enabled transition is a *time progress* update: $\forall \sigma \in S, \forall (e_i, t_i) \in \sigma(cal)$, if $\sigma(now) < t_i$, then $\forall \sigma'$ such that $\sigma \to \sigma'$, $\sigma' = \sigma[now := \min(cal)]$.
3. In all states, if the current time equals a timeout, then the only transitions enabled are calendar event updates associated with the timeout: $\forall \sigma \in S$, $\exists (e_i, t_i) \in \sigma(cal)$ such that $\sigma(now) = t_i$ implies $\forall \sigma'$ such that $\sigma \to \sigma'$, $\sigma'(now) = \sigma(now)$, $\sigma'(c_j) = \sigma(c_j)$ for all $c_j \in \sigma(cal)$ such that $c_j \neq c_i$ (recalling that by convention, $c_i = (e_i, t_i)$), and $c_i \notin \sigma'(cal)$.

From the definitions, it follows that in every state, the timeouts are never in the past, and that time is monotonic:

Lemma 1 (Future timeouts). $\forall \sigma \in S$, $(e_i, t_i) \in \sigma(cal)$, $\sigma(now) \leq t_i$.

Lemma 2 (Monotonic time). $\forall \sigma, \sigma' \in S$, *if* $\sigma \to \sigma'$, *then* $\sigma'(now) \geq \sigma(now)$.

Proofs of these two lemmas are straightforward and omitted.

In a distributed system, it is convenient to distinguish global actions and local actions. Global actions are principally interprocess communication, while local actions are those carried out by each process to update its local state and produce new messages to broadcast. While both global and local actions can both be modeled as events in a calendar automata, doing so is generally overkill and complicates the model. From the global perspective, individual processes can update their local state atomically.

Again, following Dutetre and Sorea, we associate calendar events with channels in a distributed system [6]. Specializing calendars to message passing does not lose generality since all external communication from an individual process can be abstracted as message passing. Furthermore, fault models can be abstracted to act over channels rather than processes [9]. The calendar introduces real-time constraints on when processes send and receive messages.

Assume processes are indexed from a finite set Id. A *channel* from process i to j is an ordered pair (i, j). Fix a set of messages Msg. Given a channel and a timeout, let *send* be a relation on messages sent on a channel at a given time:

$$send \subseteq Id \times Id \times \mathbb{R} \times Msg$$

So $send(i, j, t, m)$ holds iff i sends to j message m at time t. Likewise, let

$$recv \subseteq Id \times Id \times \mathbb{R} \times Msg$$

be a relation on messages received on a channel at a time, so that $recv(i, j, t, m)$ holds iff the message m received by j from i at time t.

In the absence of faults, we require that messages received were previously sent and not previously received: if $(i, j, t, m) \in recv$, then $\exists t'$ such that $(i, j, t', m) \in recv$ where $t' < t$, and $\neg \exists t''$ such that $t' < t'' < t$ and $(i, j, t'', m) = (i, j, t, m)$. (We address faults in Section 2.2.)

Then an event calendar for sending and receiving messages on channels is the union of the *send* and *recv* relations.

The event of receiving a message initiates a process to update its local transition system and generate additional messages to send. When the process is updating its local transition system, the event calendar is paused. That is, updating an event $(i, j, t, m) \in recv$ also includes updating j's transition system.

2.2 Symbolic Fault Injection: a Synchronous Kibitzer

The typical approach to modeling faults is to add new state variables to each process representing its fault state. Then a node chooses actions based on its fault state. As a simple example, we might define a node that sends a good message if it is non-faulty and a bad message otherwise. In pseudo-code using guarded commands, its definition might look like the following:

```
node:
  health: Fault_Type;
  faulty(health)     --> send(bad_msg);
  non_faulty(health) --> send(good_msg);
```

But this approach mixes the specification of a node's behavior with the fault model, an aspect of the environment. Generally, nodes do not contain state variables assigned to their faults, or use their fault-status to determine their behavior![2] The upshot is that combining faults and node state divorces the specification from its implementation.

A second difficulty with model-checking fault-tolerant systems in general is that modeling faults requires adding state and non-determinism. The minimum number of additional states that must be introduced may depend non-obviously on other aspects of the fault model, specific protocol, and system size. Such constraints lead to "meta-model" reasoning, such as the following, in which Rushby describes the number of data values that a particular protocol model must include to model the full range of Byzantine faults (defined later in this section):

> 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. It would certainly be safe to introduce additional values for this purpose, but the performance of model checking is very sensitive to the size of the state space, so there is a countervailing argument against introducing additional values. A little thought will show that .... Hence, we decide against further extension to the range of values [10].

[2] There are exceptions; for example, benign faults may be detected by a node itself (e.g., in a built-in-test).

The second problem is the most straightforward to solve. In infinite-state model-checking, we can use either the integers or the reals as the datatype for values. Fault-tolerant voting schemes, such as a majority vote or mid-value selection (see Section 3), require only equality, or a total order, respectively, to be defined for the data.

The solution to the first problem is more involved. Our solution is to introduce what we call a *synchronous kibitzer* that symbolically injects faults into the model. The kibitzer decomposes the state and transitions associated with the fault model from the system itself. For the sake of concreteness in describing the synchronous kibitzer, we introduce a particular fault model, the hybrid fault model of Thambidurai and Park [11]. This fault model distinguishes Byzantine, symmetric, and manifest faults. It applies to broadcast systems in which a process is expected to broadcast the same value to multiple receivers. A *Byzantine* (or *arbitrary*) fault is one in which a process that is intended to broadcast the same value to other processes may instead broadcast arbitrary values to different receivers (including no value or the correct value). A *symmetric* fault is one in which a process may broadcast the same, but incorrect, value to other processes. Finally, a *manifest* (or *benign*) fault is one in which a process's broadcast fault is detectable by the receivers; e.g., by performing a cyclic redundancy check (CRC) or because the value arrives outside of a predetermined window.

Define a set of fault types

$$\text{Faults} = \{none, byz, sym, man\}.$$

As in the previous section, let Id be a finite set of process indices, and let the variable

$$faults : Id \to \text{Faults}$$

range over possible mappings from processes to faults.

The hybrid fault model assumes a broadcast model of communication. A $broadcast : Id \to 2^{Id} \to \mathbb{R} \to Msg \to 2^E$ takes a sender, a set of receivers, a real-time, and a message to send each receiver, and returns a set of calendar events:

$$broadcast(i, R, t, m) = \{m | j \in R \text{ and } send(i, j, t) = m\}$$

With this machinery, we can define the semantics of faults by constraining the relationship between a message broadcast and the values received by the recipients. For a nonfaulty process that broadcasts, every recipient receives the sent message, and for symmetric faults, there is no requirement that the messages sent are the ones received, only that every recipient receives the same value:

$$
\begin{array}{l}
nonfaulty\_constraint = \\
\quad \forall i, j \in Id, t \in \mathbb{R}. \\
\qquad faults(i) = none \\
\qquad \text{implies } recv(i,j,t) = send(i,j,t)
\end{array}
\quad\Big|\quad
\begin{array}{l}
sym\_constraint = \\
\quad \forall i, j, k \in Id, t \in \mathbb{R}. \\
\qquad (\quad faults(i) = sym \\
\qquad\quad \text{and } broadcast(i, \{j,k\}, t, m)) \\
\qquad \text{implies } recv(i,j,t) = recv(i,k,t)
\end{array}
$$

Byzantine faults are left completely unconstrained.

Thus, faults can be modeled solely in terms of their effects on sending and receiving messages. A node's specification does not have to depend on its fault status directly.

If the $faults$ mapping is a constant, then faults are permanently but non-deterministically assigned to nodes. However, we can easily model *transient faults* in which nodes are faulty temporarily by making $faults$ a state variable that is updated non-deterministically. Whether we model permanent or transient faults, a *maximum fault assumption* (MFA) describes the maximum number of faults permitted in the system. The $faults$ mapping can be non-deterministically updated during execution while satisfying the MFA using a constraint such as $faults \in \{f \mid mfa(f)\}$, where the MFA is defined by the function mfa.

2.3 Abstract Transition Systems

Due to the sheer size of implementation-level models, manually examining counterexamples is tedious. To scale up verification, we use abstract transition systems (also known as disjunctive invariants) [12, 13]. In this context, an abstract transition system, relative to a given transition system $\mathcal{M} = (S, I, \rightarrow)$, is a set of state predicates $A_1, \ldots, A_n$ over S and a transition system $\mathcal{M}_* = (S_*, I_*, \rightsquigarrow)$ such that:

1. $S_* = \{a_1, \ldots, a_n\}$ is a set of "abstract states" which correspond one-to-one with the state predicates A_i.
2. $\forall s \in I,\ \exists i : a_i \in I_* \wedge A_i(s)$
3. $\forall a_i \in S_* \forall \sigma, \sigma' : A_i(\sigma) \wedge \sigma \rightarrow \sigma' \implies A_{k_1}(\sigma') \vee \ldots \vee A_{k_m}(\sigma')$ where $\{a_{k_1}, \ldots, a_{k_m}\}$ are the abstract states to which $\mathcal{M}_*$ may transition from a_i.

For verification purposes, it is important to note that if $\mathcal{M}$ and $\mathcal{M}_*$ satisfy the requirements above, then $A_1 \vee \ldots \vee A_n$ is an inductive invariant of $\mathcal{M}$. We may use such an invariant freely as a powerful assumption in the proof of other invariants (see Section 3.3).

The use of abstract transition systems not only allows us to scale proofs farther, but also to improve traceability and debugging while developing a model. In models like the ones described in Section 4.2 where there are on the order of 100 state variables and counterexample traces could be 30 steps long, the designer can be easily lost trying to identify the essence. In such cases, the

values of the abstract predicates can serve to focus the designer's attention on one particular mode of the system where the counter example is taking place. At the present we do not have a good method for synthesizing the predicates $A_1, \ldots, A_n$ automatically for general systems; they must be supplied by the user.

3 Modeling and Verification for Oral Messages

The Hybrid Oral Messages (OMH) algorithm [3] is a variant of the classic Oral Messages (OM(m)) algorithm [14], originally developed by Thambidurai and Park [11] to achieve distributed consensus in the presence of a hybrid fault model. However, OMH had a bug, as originally formulated, which was corrected and the mended algorithm was formally verified by Lincoln and Rushby using interactive theorem-proving [3].

First, we briefly describe the algorithm, sketch our instantiation of the model for the particular protocol in Section 2, then describe it's invariants.

3.1 OMH(m) Algorithm

OMH is a recursive algorithm that proceeds in rounds of communication. Here we give a recursive specification for OMH(m), parameterized by the number of rounds, m. Consider a finite set of nodes N. Distinguish one node as the *general*, g, and the remaining nodes $L = N \setminus \{g\}$ as the *lieutenants*. We assume the identity of any general or lieutenant cannot be spoofed. Broadcast communication proceeds in rounds. Denote any message that is detectably faulty (e.g., fails a CRC) or is absent, by *ERR*. Additionally, in the algorithm, nodes report on values they have previously received. In doing so, nodes must differentiate *reporting ERR* from an *ERR* itself. Let R denote that an error is being reported. Finally, let V be a special, designated value.

The algorithm is recursively defined for $m \geq 0$:

- OMH(0): g broadcasts a value to each lieutenant and the lieutenants return the value received (or *ERR*).
- OMH(m), $m > 0$:
  1. g broadcasts a value to each lieutenant, l.
  2. Let l_v be the value received by $l \in L$ from g. Then for each l, execute OMH($m-1$), assigning l to be the general and $L \setminus \{l\}$ to be the lieutenants. l sends l_v, or R if $l_v = ERR$.
  3. For each lieutenant $l \in L$, remove all *ERR* values received in Step 2 from executing OMH($m-1$). Compute the majority value over the remaining values, or V if there is no majority. If the majority value is R, return E.

In particular, OMH(1) includes two rounds of broadcast communication: one in which the general broadcasts, and one in which the lieutenants exchange their values.

OMH is designed to ensure *validity* and *agreement* properties under suitable hypotheses on the number and type of faults in the system. Validity states that

if the general is nonfaulty, then every lieutenant outputs the value sent by the general. Agreement states that each lieutenant outputs the same value. More formally, Let l_i, l_j denote the outputs of lieutenants $i, j \in L$, respectively, and let v be the value the general broadcasts:

$$\forall i. \quad l_i = v \text{ (Validity)} \qquad \forall i, j. \quad l_i = l_j \text{ (Agreement)}$$

We described a hybrid fault model in Section 2.2. Under that fault model, validity and agreement hold if $2a+2s+b+1 \geq n$, where n is the total number of nodes, a is the number of Byzantine (or asymmetric) faults, s is the number of symmetric faults, and b is the number of benign faults. Additionally, the number of rounds m must be greater or equal to the number of Byzantine faults, a [3,11].

3.2 Model Sketch

We have implemented OMH(1) (as well as the variants described in Section 4.2) in the Symbolic Analysis Laboratory (SAL) [15]. SAL contains a suite of model-checkers. In our work, we use infinite-state (SMT-based) k-induction [6].

We follow Rushby [10] in "unrolling" the communication among lieutenants into two sets of logical nodes: *relays* and *receivers*. Relays encode the lieutenants' Step 2 of the OMH algorithm, in which they rebroadcast the values received from the general after filtering manifestly bad messages, while the receivers encode the voting step. We refer to the general as the *source*. The unrolling shows that a generalization of the original algorithm holds: the number of relays and receivers need not be the same. We model communication through one-way, typed channels. The source broadcasts a message to each relay which, in turn, each broadcast their messages to all receivers.

The relays and receivers explicitly send and receive messages and store them in local buffers as needed. In addition, the receivers implement the Fast MJRY algorithm [7].

Our SAL model defines seven transition systems in total: `clock`, `source`, `relay` (parametrized over an ID), `receiver` (parametrized over an ID), `observer`, `abstractor`, and `abstract_monitor`. The first four of these are composed asynchronously, in an intermediate system we label `system`, and share access to a global calendar consisting of event slots (message, time), one for each channel in the system. The `clock` transition system is responsible for updating a global variable `t` (called *now* in Section 2.1) representing time according to the rules for calendar automata.

The asynchronous composition of the system relaxes the original specification of the algorithm considerably. For example, in our implementation, a receiver may receive a message from one relay before another relay has received a message from the source. We only require that all relays and receivers have executed before voting. With a general asynchronous model, it is easy to refine it further; for example, we refine it to a time-triggered model in Section 4.2.

The `observer` is a synchronous observer [16] that encodes the validity and agreement properties as synchronously-composed transition systems. State variables denoting validity and agreement are set to be false if the receivers have completed their vote but the respective properties do not hold.

Finally, the `abstractor` and `abstract_monitor` encode an abstract transition system for the system, as described in Section 2.3.

3.3 Invariants

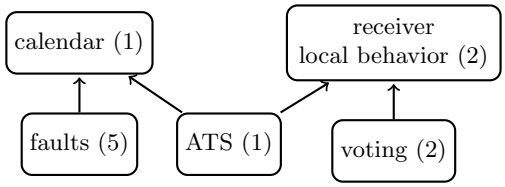

Fig. 1. Invariant classification and dependencies.

To make the proof scalable, we specify inductive invariants to be used by SAL's k-induction engine. There are 11 invariants, falling into five categories:

1. *Calendar automata*: Lemmas relating to the calendar automata model. These include lemmas such as time being monotonic, channels missing messages if there is no calendar event, and only nodes associated with a calendar event may execute their local transition systems.
2. *Abstract transition sytem (ATS)*: Lemmas relating the ATS states to the implementation states.
3. *Receiver local behavior*: Lemmas describing the modes of behavior of the receivers. The major modes of their behavior are receiving messages, then once it has filled its buffer, it votes, and after voting returns the result. An additional lemma notes that the messages currently received plus missing messages equals the total number of expected messages.
4. *Faults*: Lemmas characterizing the effect of a fault in a single broadcast. Examples include lemmas stating that if a node receives a faulty message, some "upstream" node in the communication path was faulty. Another example is that the faults of messages latched by a node in its buffer match the faults ascribed to the sender in the calendar event.
5. *Voting*: Lemmas proving that the Fast MJRTY algorithm implements a majority vote, if one exists. These lemmas are nearly verbatim transcriptions from the journal proofs for the algorithm [7].

The proof structure is shown in Figure 1. The number of lemmas per category are shown in parentheses. Arrows denote dependencies. For example, the ATS lemmas depend on both the calendar automata and receiver state-machine lemmas. As can be seen, the proof structure is modular. The calendar lemmas are

general and independent of any particular protocol or fault model. Similarly, lemmas about the internal behavior of a receiver is independent of the global protocol behavior. It is also independent of the effect of faults on the system—the only "knowledge" of faults that receiver has is whether a fault is benign or not. Lemmas about the behavior of faults in the system are also independent of the particular protocol being modeled. Likewise, lemmas about the particular voting algorithm used depend only on the receiver's internal behavior. Only the ATS depends on both calendar-specific and local state-machine results, since it is an abstraction of the entire system implementation. Recall, however, that the ATS is a convenience for debugging and can be elided.

4 Experimental Results

Here we present two classes of experimental results. First, we demonstrate the scalability results of the verification, despite the low-level modeling. Then we describe modularity results, demonstrated by making modifications to the model and re-validating the model.

4.1 Scalability

Relays \ Receivers	1	2	3	4	5	6	7	8	9	10
1	7	9	12	15	21	25	32	40	54	74
2	17	14	21	30	42	53	74	99	144	-
3	21	22	40	50	81	102	155	279	-	-
4	27	34	59	99	141	237	1114	-	-	-
5	22	94	125	335	**T**	1406	-	-	-	-
6	36	132	2966	844	2457	-	-	-	-	-
7	83	487	**T**	**T**	-	-	-	-	-	-
8	298	**T**	**T**	-	-	-	-	-	-	-
9	1428	**T**	-	-	-	-	-	-	-	-
10	**T**	-	-	-	-	-	-	-	-	-

Fig. 2. Benchmark of full proof computation time for OMH(1) implementation. Times are in seconds with a timeout (**T**) limit of one hour. Dashes ('-') denote no benchmark was run.

We present benchmarks in Figure 2. The benchmarks were performed on a server with Intel Xeon E312xx (Sandy Bridge) CPUs. The table provides execution times in seconds, with a timeout limit of one hour, for verifying the model, given a selected number of relays and receivers. The voting logic is in the receivers, so they have substantially more state than the relays, and dominate the execution time. The execution times sums the execution times for verifying each of the eleven lemmas individually, as well as the final agreement and validity

theorems. Each proof incurs the full startup, parsing, type-checking, and model-generation time of SAL. Observe the theorems hold even in the degenerate cases of one relay or one receiver.

As a point of comparison, Rushby presents an elegant high-level model of OM(1), also in SAL [10]. For small numbers of relays/receivers, the verification of Rushby's model is much faster, likely due to making only one call to SAL. However, for six relays and two receivers, it takes 449 seconds and timeouts (at one hour) for seven relays and two receivers. Checking Rushby's model requires use of symbolic, BDD-based model-checking techniques which are well-known to scale poorly. On the other hand, our model requires the use of k-induction which scales well, but requires (inductive) invariants to be provided.

4.2 Modular Verification

	Transition systems (7 total)	Definitions (58 total)	Invariants (11 total)	Invariant classes (5 total)
Omissive Asymmetric Faults	none	1 new, 2 modified	2 modified	faults
Time-Triggered Messaging	source, relays, receivers, ATS	3 new	2 modified, 3 modified	calendar, faults
Mid-Value Selection	receivers	4 new, 3 modified	2 modified	ATS, voting

Fig. 3. Refactoring effort for protocol modifications, measured by which portions of the model have to be modified.

To demonstrate the modularity of the modelling and verification approach, in this section, we explore variants to the model and report the effort required to implement the modifications and repair the proofs. The results are summarized in Figure 3 and sketched below. In the table, for each modification, we report how much of the model must be modified. We report on four aspects of the system: which transition systems are modified (as described in Section 3.2), how many definitions have to be added or modified, the number of invariants that have to be added or modified, and which invariant classes (as defined in Section 3.3) those lemmas belong to. We modify the implementation along the axes of faults, time, and local node behavior.

Omissive Asymmetric Faults. Removing faults already described by the fault model is easy. Recall that in our model faults do not appear in the system specification and only operate on the calendar. Removing a fault from the system requires only setting the number of a particular kind of faults to zero in the maximum fault assumption.

Adding new kinds of faults requires more work but is still modular. Consider adding *omissive asymmetric faults*, a restriction of Byzantine faults in which a broadcaster either sends the correct value or a benign fault [8], to the fault model. Doing so requires modifying none of transition systems, because of the synchronous kibitzer. We add a new uninterpreted function definition for omissive asymmetric faults, then modify the type of faults, and their effect on the calendar. Two invariants, both in the class of invariants cover faults, are extended to cover the cases where a sender is omissive asymmetric.

Time-Triggered Messaging. A *time-triggered* distributed system is one in which nodes are independently clocked, but clock constraints allow the model to appear as if it is executing synchronously [17].

Changing the model to be time-triggered principally requires making the source, relays, and receivers driven explicitly by the passage of time (we do not model clock drift or skew). As well, a "receive window" is defined at which messages from non-faulty nodes should be received. Messages received outside the window are marked as coming from manifest-faulty senders. The model requires three new definitions to encode nondeterministic message delay and two are small helper functions. The guards in the relays and receivers are modified to latch messages received outside the receive windows as being manifest faults. The ATS definition is modified to track the times in the calendar, not just the messages. Two new calendar invariants are introduced, stating that the calendar messages are either empty, or their time-stamps fall within the respective message windows. Then, three invariants classifying faults are relaxed to allow for the possibility of faulty nodes sending benign messages.

Mid-Value Selection. Our OMH(1) model leverages a majority vote in order to tolerate faults. Another choice for the fault masking algorithm used is mid-value selection. This choice is common in applications involving hardware, signal selection, or cases where information about congruence is useful. To implement mid-value selection in our model, we allow messages sent to take values in $\mathbb{R}$ and the receiver transition system is modified in two ways. First, a second buffer is introduced which will hold the sorted contents of the main buffer once voting has commenced. Second, a mid-value select function is called on the sorted buffer and the result is stored as the receiver's vote. The only invariants needing modification were the ATS definition (to account for the values stored by the new buffer and the relation between it and the main buffer) and the voting invariant.

4.3 Proof Effort Remarks

The lemmas described in Section 3.3 are constructed by-hand and represent multiple days of effort, but that effort includes both model and protocol construction and generalization as well as verification. The counterexamples returned by SAL are very useful for strengthening invariants, but tedious to analyze—a model with five relays and two receivers contains 90 state variables, and there are known

counterexamples to models that size [3]. Once we developed the synchronously-composed ATS observer, the verification effort was sped up considerably.

The invariants are surprisingly modular. One benefit of a model-checking based approach is that it is automated to rerun a proof of a theorem omitting lemmas to see if the proof still holds. This allowed us to explore reducing dependencies between invariants related to different aspects of the system.

The modifications to the implementation described in Section 4.2 took at most hours to develop. Moreover, most of the invariants do not concern the specific protocol modeled at all, and we hypothesize that for completely different fault-tolerance protocols, only the modeling aspects related to the protocol behavior and local node behavior would change, and the invariant structure would remain modular.

Moreover, we are agnostic about how lemmas are discovered. As techniques like IC3 scale, they may be discovered automatically. k-induction in infinite-state model-checking blurs the lines between interactive and automated theorem proving. IC3 can even be strengthened using k-induciton [18].

5 Related Work

The Oral Messages algorithm and its variants and its variants have a long history of formal verification. OM(1) was verified in both the PVS and ACL2 interactive theorem-provers [2]. Also in ACL2, an implementation of a circuit design to implement OM(1) is given [1]; the low-level model most closely relates to our level of detail. A refinement-based verification approach is used, and OM(1) is specialized to a fixed number of nodes. Bokor *et al.* describe a message-passing model for synchronous distributed algorithms that is particularly amenable to partial-order reduction for explicit-state model-checking [19]. The model is efficient for up to five nodes, but results are not presented beyond that. Very recently, Jovanović and Dutertre use a "flattened" high-level model of OM(1) as a benchmark for IC3 augmented with k-induction [18].

Moreover, our work is heavily influenced by previous verifications of fault-tolerant and real-time systems in SAL [6, 10, 13].

6 Conclusions

This work fits within a larger project, in collaboration with Honeywell Labs, to build an *architectural domain-specific language* (ADSL) for specifying and verifying distributed fault-tolerant systems. The ADSL should be able to synthesize both software and/or hardware implementations as well as formal models for verification. Before building such an ADSL, we needed a scalable general formal model to which to compile, leading to the work presented in this paper. We hypothesize that the ADSL will make refactoring even easier, and we can generate invariants or invariant templates useful for verification. Indeed, we have

developed a preliminary ADSL that generates C code as well as formal models in SRI's Sally [18], to be described in a future paper.[3]

Beyond building an ADSL, another avenue of research is producing a formal proof that a software implementation satisfies the node specification in our formal model. While our model of node behavior is low-level, there are gaps. For example, our work is in SAL's language of guarded commands [15] and needs to be either refined or verified to be equivalent to a software implementation's semantics. Another aspect is that behavior related to networking, serialization, etc. is left abstract, implicit in the *send* and *recv* functions.

Acknowledgments

This work is partially supported by NASA contract #NNL14AA08C. We are indebted to our collaborators Brendan Hall and Srivatsan Varadarajan at Honeywell Labs, and to Wilfredo Torres-Pomales at NASA Langley for their discussions and insights. Additionally, we acknowledge that this work is heavily inspired by a series of papers authored by John Rushby.

References

1. W. R. Bevier and W. D. Young, "The proof of correctness of a fault-tolerant circuit design," Computational Logic, Inc., Tech. Rep. 57, 1990, available at http://computationallogic.com/reports/index.html.
2. W. D. Young, "Comparing verification systems: Interactive Consistency in ACL2," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 214–223, Apr. 1997.
3. P. Lincoln and J. Rushby, "A formally verified algorithm for interactive consistency under a hybrid fault model," in *In Fault Tolerant Computing Symposium 23*. IEEE Computer Society, 1993, pp. 402–411.
4. S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, 1995.
5. T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 2007, pp. 398–407.
6. B. Dutertre and M. Sorea, "Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. LNCS, vol. 3253. Springer, Sep. 2004.
7. R. S. Boyer and J. S. Moore, *MJRTY—A Fast Majority Vote Algorithm*. Springer, 1991, pp. 105–117.
8. M. H. Azadmanesh and R. M. Kieckhafer, "Exploiting omissive faults in synchronous approximate agreement," *IEEE Trans. Computers*, vol. 49, no. 10, pp. 1031–1042, 2000.
9. L. Pike, J. Maddalon, P. Miner, and A. Geser, "Abstractions for fault-tolerant distributed system verification," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, vol. 3223. Springer, 2004, pp. 257–270.

[3] https://github.com/GaloisInc/atom-sally

10. J. Rushby, "SAL tutorial: Analyzing the fault-tolerant algorithm OM(1)," Computer Science Laboratory, SRI International, Menlo Park, CA, CSL Technical Note, Apr. 2004, available at http://www.csl.sri.com/users/rushby/abstracts/om1.
11. P. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in *Symposium on Reliable Distributed Systems*. IEEE, 1988, pp. 93–100.
12. J. Rushby, "Verification diagrams revisited: Disjunctive invariants for easy verification," in *Computer-Aided Verification, CAV '2000*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Chicago, IL: Springer-Verlag, Jul. 2000, pp. 508–520.
13. B. Dutertre and M. Sorea, "Timed systems in SAL," SRI International, Menlo Park, CA, SDL Technical Report SRI-SDL-04-03, July 2004.
14. L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.
15. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muoz, S. Owre, H. Rue, J. Rushby, V. Rusu, H. Sadi, N. Shankar, E. Singerman, and A. Tiwari, "An overview of SAL," in *NASA Langley Formal Methods Workshop*, 2000, pp. 187–196.
16. J. Rushby, "The versatile synchronous observer," in *Specification, Algebra, and Software, A Festschrift Symposium in Honor of Kokichi Futatsugi*, vol. 8373. Springer, Apr. 2014, pp. 110–128.
17. H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.
18. D. Javanović and B. Dutertre, "Property-directed k-induction," in *Formal Methods in Computer Aided Design (FMCAD)*, 2016.
19. P. Bokor, M. Serafini, and N. Suri, *On Efficient Models for Model Checking Message-Passing Distributed Protocols*. Springer, 2010, pp. 216–223.