SRI International

CSL Technical Note • April 2004

SAL Tutorial:
Analyzing the Fault-Tolerant Algorithm OM(1)

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

This research was supported by NASA Langley Research Center under contract NAS1-20334 and Cooperative Agreement NCC-1-377 with Honeywell Tucson.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

The resources of SAL allow many kinds of systems to be modeled and analyzed. However, it requires skill and experience to exploit the capabilities of SAL to the best effect in any given problem domain. This tutorial provides an introduction to the use of SAL in modeling and analyzing fault-tolerant systems.

The example considered here is a simple variant on the classical one-round Oral Messages algorithm OM(1) for Byzantine agreement and will be familiar to many computer scientists. The SAL model developed here is available for download, so that users can repeat the analyses described, and exercises are suggested for additional experiments.

Contents

1 Interactive Consistency and the Basic Algorithm 1

2 Modeling in SAL 2

3 Exercises 18

References 20

A Hints for Exercises 23

1 Interactive Consistency and the Basic Algorithm

Interactive Consistency (also known as source congruence and Byzantine agreement) is the problem of transferring a value from a single *source* to multiple *receivers* in a way that guarantees certain properties, even in the presence of faults [LSP82, PSL80]. One desired property is *agreement*: all nonfaulty receivers should get the same value. The trivial algorithm that simply sends the value directly from the source to each receiver cannot guarantee agreement when the source is faulty (since a faulty source could send different values to different receivers). To overcome this problem, we arrange that the source sends its value to a set of *relays*; each relay then sends the value it obtained to each of the receivers, and each receiver takes a majority vote over the values it obtains from the relays (see Figure 1). This algorithm is an "unrolled" variant of the one-round version of the classical *Oral Messages* algorithm [LSP82], and it is also similar to an earlier algorithm of Davies and Wakerly [DW78].

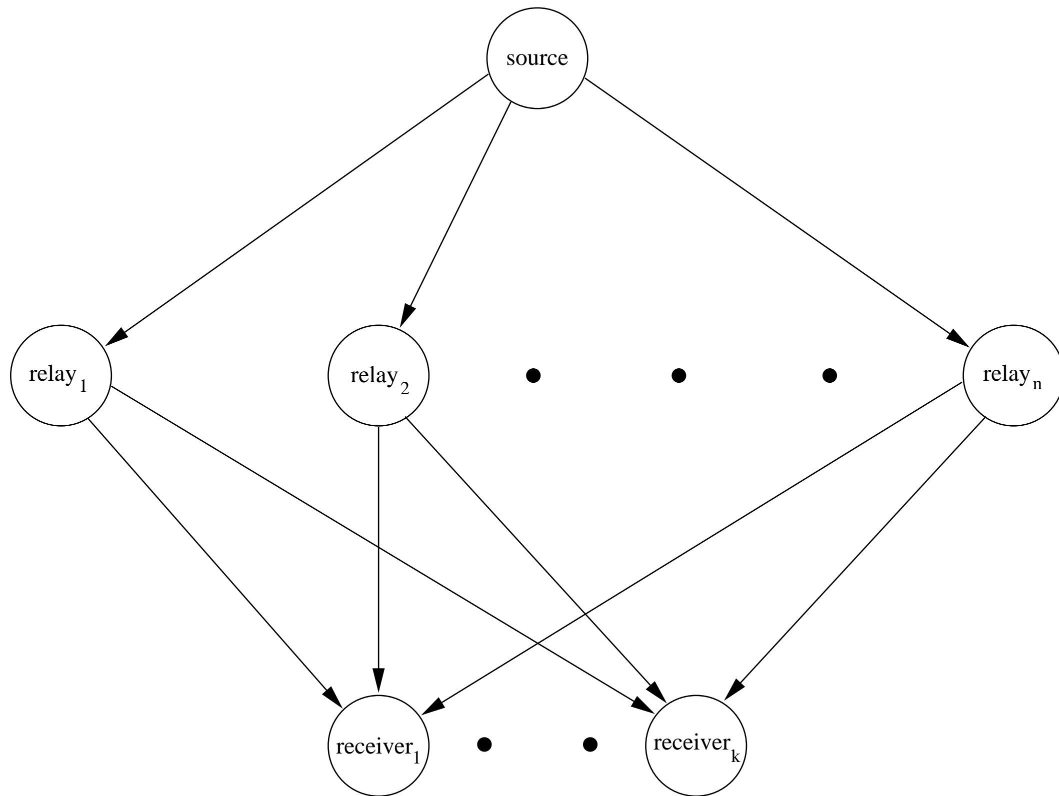

Figure 1: OM(1) Algorithm

Assuming there are at least three relays, and none of them is faulty, then it is easy to see that this algorithm ensures agreement even if the source is faulty. Of course, introduction

of the relays means there are more components that can be faulty, and faulty relays can corrupt the value sent by a nonfaulty source. Thus, a second desired property is *validity*: when the source is nonfaulty, each nonfaulty receiver should obtain the value actually sent. If we have three relays and at most one of them is faulty, then it is easy to see that validity is guaranteed. If we have two faulty relays, however, then validity is not guaranteed with only three relays, but it is with five. But even with five relays and only one of them faulty, agreement is not guaranteed if the source is faulty. It is clear that the number of faults that can be tolerated is related to the number of relays and our task is to use the model checkers of SAL to help formulate and verify candidate relationships.

2 Modeling in SAL

Prerequisites for this tutorial are to have SAL 2 installed on your machine and to have the SAL language manual and other SAL documentation available. All these can be obtained from `http://sal.csl.sri.com`.

The SAL language [dMOS01] provides notations for specifying state machines and their properties, while the SAL system [dMOR+04] provides model checkers and other tools for analyzing properties of state machine specifications written in SAL. The basic unit of specification in SAL is a *module*. A module can directly specify a state machine, or it can specify the composition of other modules. Modules can be composed either synchronously (meaning they all operate in lockstep) or asynchronously (meaning that exactly one module makes a move at each step).

In our example, it is natural to propose specifying the source and each of the relays and receivers as a separate module, and we then need to consider the type of composition to be employed. The Oral Messages algorithm is actually a synchronous algorithm [Lyn96] and, although the term "synchronous" has different meanings in distributed algorithms and formal methods, it is synchronous composition that is appropriate here. We think of the algorithm as proceeding in synchronized *stages*: first the source sends out its value, then the relays collectively pass on the values obtained, and then the receivers perform their majority vote. To coordinate this staging, we use a controller module, which plays a rôle similar to the clock in a synchronous hardware implementation. We number the stages 1, 2, 3, and the task of the controller will be to output a variable called `pc` (for "program counter") that tells the other modules what the current stage is.

Now a SAL module describes a state machine by specifying a transition relation on its state variables and typically does so by defining the "new" (i.e., post-transition) values of its local and output variables in terms of the "old" (i.e., pre-transition) values of its input, local, and output variables (it is also possible to use new values here, provided there are no circularities). These descriptions are generally stated as guarded commands, where the guards trigger appropriate transitions. The values referenced in the guards are typically old ones (though here again, it is possible to use new values). If we decide that our specifications will reference the old value of `pc` in their guards, then at the time this equals 1 (and

the source module should be active), the controller will already be setting the new value to 2. Since we want to examine the outputs of the `receiver` modules (which operate at stage 3) *after* they have performed their computation, we need an additional stage (i.e., 4) for this. If we allow the controller to count up to stage 4 and then remain there, we will need to take care that the outputs of the receivers are latched. While this is perfectly feasible, it is easier to allow `pc` to progress to the value 5 and stay there. Thus, we have derived the following initial specification for the controller module.

Preliminary

```
om1: CONTEXT =
BEGIN

stage: TYPE = [1..5];

controller: MODULE =
BEGIN
OUTPUT
  pc: stage
INITIALIZATION
  pc = 1;
TRANSITION
[
  pc <= 4 --> pc' = pc+1
[]
  ELSE -->
]
END;
```

The explicit `ELSE` prevents the module deadlocking when the guard is not satisfied: because there is no command associated with this guard, the module simply stutters (i.e., leaves all its state variables unchanged).

Next, we can introduce `n` as the number of relays, and `relays` as the type that indexes these, and similarly `k` as the number of receivers and their corresponding index type. We set `n` and `k` to small values for our initial experiments.

```
n: NATURAL = 3;
k: NATURAL = 2;

relays: TYPE = [1..n];
receivers: TYPE = [1..k];
```

We now need to think about the values that the source, relays, and receivers will operate on. We need a "correct" value, and some number of incorrect values that can be introduced by faulty sources and relays. 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 the way faulty relays have their most significant impact is by tipping the majority vote in a receiver one way or the other, and this can only be achieved if they use the same values as nonfaulty relays. Hence, we decide against further extension to the range of values. We will however, need a special value to indicate a missing or corrupted message. It is convenient to use 0 for this missing value, then 1...`n` for the incorrect values, and `n+1` for the correct value. We therefore arrive at the following declarations.

```
vals: TYPE = [0..n+1];
missing_v: vals = 0;
correct_v: vals = n+1;
```

We can now give a preliminary specification for the `source` module. This is activated when the `pc` is 1, and sends a value to each of the relays. We use an array `s_out` as the collection of values sent to the relays. In this first version of the specification, we ignore faulty behavior.

Preliminary

```
rvec: TYPE = ARRAY relays OF vals;

source: MODULE =
BEGIN
OUTPUT
  s_out: rvec
INPUT
  pc: stage
TRANSITION
[
  pc = 1 --> s_out' = [[i:relays] correct_v]
[]
  ELSE -->
]
END;
```

This module sends values to the output array `s_out` only when `pc` is 1, otherwise it leaves its state variables unchanged. The value `[[i:relays] correct_v]` assigned to `s_out` is an array literal: it specifies an array whose index type is `relays` and whose value is `correct_v` everywhere.

We can give a similar specification for the relays in the fault-free case. The `relay` module is parameterized by the value `i` and specifies the behavior of the `i`'th relay. The relay is active only when the `pc` is 2, in which case it simply copies the value it received from the source `r_in` to its array of outputs (one entry for each receiver) `r_out`.

```
vvec: TYPE = ARRAY receivers OF vals;                    Preliminary

relay[i: relays]: MODULE =
BEGIN
INPUT
  pc: stage,
  r_in: vals
OUTPUT
  r_out: vvec
TRANSITION
[
  pc = 2 --> r_out' = [[p:receivers] r_in]
[]
  ELSE -->
]
END;
```

Although we have not yet specified the receiver module, we can begin to assemble the modules we have so far into a system. We want the synchronous composition of the `controller` and `source` modules, together with an `n`-fold synchronous composition of `relay` modules.

```
system: MODULE =                                         Preliminary
   controller
     || source
     || (|| (i:relays): relay[i]);
```

One problem with this construction is that each instance of the `relay` module is driving the output `r_out`, whereas only a single module is allowed to drive a variable declared as an output. This issue is always present in synchronous constructions, and the solution is to introduce an array `vecs` of variables and to assign the output of each module to a separate element of the array. Hence we arrive at the following construction.

```
system: MODULE =                                         Preliminary
   controller
     || source
     || (WITH OUTPUT vecs: ARRAY relays OF vvec
          (|| (i:relays): RENAME r_out TO vecs[i] IN relay[i]));
```

This composition is legal, but fails to “wire up” input and output variables correctly: the output of the `source` module is `s_out`, while the input of the `relay` modules is `r_in`. Furthermore, each `relay` needs to connect its `r_in` to the appropriate element of the `s_out` array. We therefore arrive at the following construction.

```
system: MODULE =
   controller
     || source
     || (WITH OUTPUT vecs: ARRAY relays OF vvec
          WITH INPUT s_out: rvec
            (|| (i:relays): RENAME r_in TO s_out[i],
                                   r_out TO vecs[i]
                            IN relay[i]));
```

Preliminary

Now that we have the basic structure in place, we can consider the specification of the `receiver` modules, and the modeling of faults. Because the purpose of the receivers is to mask faults through majority voting, it seems best to turn first to the modeling of faults.

We need to decide what kinds of faults should be modeled, and where and how they should be introduced into the specification. In analyzing algorithms for interactive consistency, it is conventional to associate faults with the processors on which the algorithms run (though more elaborate treatments also associate faults with communication links); in our model, this would correspond to associating faults with the `source` and `relay` modules. For the `source` and each `relay` module, we could add a state variable that indicates whether that module is faulty; if it is, then the behavior of the module will be adjusted in some way. To model all possible combinations of faults, we would nondeterministically assign values to the state variables that determine faultiness during initialization. These "faultiness" variables could be local to each module, but it will be easier to control the patterns of faults if they set in some central place. Also we will need to count how many faulty modules are present (because the properties we want to model check will be of the form "validity is achieved provided there are not too many faults") and this will most easily be achieved if the "faultiness" variables are the outputs of some module. These considerations lead to the choice that the "faultiness" variables will be outputs of the `controller` module, and inputs to `source` and `relay` modules.

In the earliest treatments of interactive consistency, all faults were considered equal and results were stated in forms such as "to withstand r faults, at least $3r + 1$ nodes and r rounds are required" [PSL80]. However, the early papers were also the first to identify and consider the "worst possible," or *Byzantine* kinds of faults—namely those that behave inconsistently (e.g., sending different values to different receivers) [LSP82, PSL80]. But although those papers gave plausible descriptions of "Byzantine" behavior, their analysis did not rely on these intuitions—for they were conducted with *no assumptions* about the behavior of faulty components. Formal treatments of these analyses undertaken with theorem provers similarly used no axioms about the behavior of faults [Rus92, BY92, You97]. In model checking, on the other hand, we have to assign explicit behaviors to the faulty components. The closest we can come to Byzantine behavior is to allow faulty modules to make nondeterministic assignments to state variables. Nondeterministic assignments in SAL are specified by the `IN` construct (this is not the same `IN` as used in specifying the `system` composition above), so that whereas the fully deterministic assignment to `s_out` by a nonfaulty `source` is given by

```
s_out' = [[i:relays] correct_v]
```
Example

the fully nondeterministic assignment by a faulty `source` is given as follows.

```
s_out' IN {a: rvec | TRUE}
```
Example

Now, fully Byzantine behavior, as modeled by totally nondeterministic assignments, poses a difficult challenge, and no algorithm can tolerate more than a third of its components delivering this kind of behavior, so we may also be interested in less demanding kinds of faults. An interesting fault model of this kind is the *hybrid* model introduced by Thambidurai and Park [TP88]: in addition to *arbitrary* (i.e., Byzantine) faults, they consider *manifest* and *symmetric* faults. A manifest fault is one that is reliably detected by all nonfaulty components (e.g., a missing message or one with an incorrect checksum), while a symmetric fault is one that is not detectable (i.e., it is a wrong rather than invalid or missing value) but that has the *same* manifestations to all receivers. The hybrid fault model is attractive in the larger context for which the example developed here was originally developed because that context is concerned with systems running on the Time Triggered Architecture (TTA) [KB03], where Byzantine faults are strongly contained [BKS02] and message loss (i.e., manifest faults) is the main concern.

We therefore modify our previous specification for the `controller` module by introducing `faults` as an enumerated type and output variables `sf` and `rf` that indicate the type of fault afflicting the `source` or `relay` modules, respectively; the fault type `none` is used to indicate a nonfaulty module. To explore all possible fault configurations, we use fully nondeterministic assignments to initialize these variables.

```
faults: TYPE = {arbitrary, symmetric, manifest, none};

controller: MODULE =
BEGIN
OUTPUT
  pc: stage,
  sf: faults,
  rf: ARRAY relays OF faults
INITIALIZATION
  pc = 1;
  sf IN {v: faults | TRUE};
  rf IN {a: ARRAY relays OF faults | TRUE};
TRANSITION
[
  pc <= 4 --> pc' = pc+1
[]
  ELSE -->
]
END;
```
Preliminary

This fault treatment is adequate, but perhaps a little *too* nondeterministic. It is well known, and easy to check, that a one-round algorithm such as this cannot tolerate a Byzantine relay when the source is also Byzantine. We will reduce the statespace if we refine the specification of `rf` to eliminate the `arbitrary` choice when `sf` has already been assigned the `arbitrary` value. Hence, we derive the following final specification for the controller.

```
controller: MODULE =
BEGIN
OUTPUT
  pc: stage,
  sf: faults,
  rf: ARRAY relays OF faults
INITIALIZATION
  pc = 1;
  sf IN {v: faults | TRUE};
  rf IN {a: ARRAY relays OF faults |
      FORALL (i:relays):  sf = arbitrary => a[i] /= arbitrary };
TRANSITION
[
  pc <= 4 --> pc' = pc+1
[]
  ELSE -->
]
END;
```

The source module can now be changed to the following specification.

```
source: MODULE =
BEGIN
OUTPUT
  s_out: rvec
INPUT
  pc: stage,
  sf: faults
TRANSITION
[
  pc = 1 AND (sf = none OR sf = symmetric) -->
    s_out' = [[i:relays] correct_v]
[]
  pc = 1 AND sf = manifest -->
    s_out' = [[i:relays] missing_v]
[]
  pc = 1 AND sf = arbitrary -->
    s_out' IN {a: rvec | TRUE}
[]
  ELSE -->
]
END;
```

A symmetric fault has no useful interpretation for the source module, so we treat it the same as the nonfaulty case. For the manifest case, we send the special missing_v value to each of the relays and for the arbitrary case, we choose nondeterministic values. Notice that because missing_v and correct_v are elements of the type vals, it is possible for an arbitrary faulty module to send bad values to some relays, missing values to others, and the correct value to still others.

The `relay` modules are elaborated in a similar manner to model faulty behavior.

```
relay[i: relays]: MODULE =
BEGIN
INPUT
  pc: stage,
  r_in: vals,
  rf:  faults
OUTPUT
  r_out: vvec
TRANSITION
[
  pc = 2 AND rf = none -->
    r_out' = [[p:receivers] r_in]
[]
  pc = 2 AND rf = manifest -->
    r_out' = [[p:receivers] missing_v]
[]
  ([] (x:vals): pc = 2 AND rf = symmetric -->
    r_out' = [[p:receivers] x])
[]
  pc = 2 AND rf = arbitrary -->
    r_out' IN {a: vvec | TRUE}
[]
  ELSE -->
]
END;
```

The novel case here is the treatment of `symmetric` faults. The idea is to chose some arbitrary value `x`, then send that to every receiver. This is specified by the `([] (x:vals): ...` *multicommand* construction, which effectively creates a separate guarded command for each value `x` in the type `vals`. SAL operates by nondeterministically choosing one command to execute from among those whose guards are true; thus, if `pc` is 2 and `rf` is `symmetric`, all instances of this command will be eligible and one will be chosen nondeterministically.

Following these changes, we need to adjust the `system` specification to connect the `rf` input variable of each `relay` module with the array output by the `controller` module.

```
system: MODULE =                                          Preliminary
   controller
     || source
     || (WITH OUTPUT vecs: ARRAY relays OF vvec
         WITH INPUT s_out: rvec
         WITH INPUT rf: ARRAY relays OF faults
           (|| (i:relays): RENAME r_in TO s_out[i],
                                  r_out TO vecs[i],
                                  rf TO rf[i]
                           IN relay[i]));
```

We are now ready to specify the `receiver` modules that take the values output by the relays and subject them to a majority vote. Our immediate challenge is to specify majority voting. There is linear-time algorithm for majority voting [BM91] that has been specified as a recursive function in PVS and this could be translated into SAL. However, in model checking we have finite domains—in particular, the type `vals` is finite—so perhaps we can exploit this to allow simpler constructions that may be easier for the model checker to interpret efficiently. One way to exploit the finite range of `vals` is simply to count the number of occurrences of each value: a value is the majority if its count times 2 is greater than `n`.

This suggests the following specification for the `receiver` module. The transition creates a guarded command for each value `i` and checks whether the number of instances of that value in the input array `inv` satisfies the condition for being the majority value; if it does, then the output variable `vote` is set to that value, otherwise (i.e., if these is no majority) the `vote` is set to some fixed value (we choose `missing_v`).

```
receiver[p:receivers]: MODULE =                           Preliminary
BEGIN
INPUT
  pc: stage,
  inv: rvec
OUTPUT
  vote: vals
TRANSITION
[
  ([] (i: vals):
     pc = 3 AND 2*count(inv, i) > n  --> vote' = i)
[]
  ELSE --> vote' = missing_v
]
END;
```

To complete this, we need to specify the function `count`. As is usual in functional programming, this is defined in terms of a recursive helper function `count_h` that uses an accumulator `acc`.

```
all: TYPE = [0..n];

count_h(a: rvec, v: vals, acc: all, i: relays): all =
   LET this_one: [0..1] = IF a[i]=v THEN 1 ELSE 0 ENDIF IN
   IF i=1 THEN acc + this_one
     ELSE count_h(a, v, acc + this_one, i-1)
   ENDIF;

count(a: rvec, v: vals): all = count_h(a, v, 0, n);
```

We have one remaining problem with the specification of the `receiver` module: the module expects an `rvec` (i.e., an `ARRAY relays OF vals`) as input, but each relay outputs a `vvec` (i.e., an `ARRAY receivers OF vals`) and these are combined in the `system` specification into `vecs`, an `ARRAY relays OF vvec`. It might seem that a `WITH INPUT...` construction could be used to align these, but the problem is that we would need to rename the element `i` of the `inv` input variable of `receiver[x]` to `vecs[i][x]`: that is to say, we need to rotate the array and SAL's `RENAME` construction does not provide for this. Hence, we need to modify the `receiver` module to take in the `vecs` value and to extract the `inv` slice locally. This is done using a `DEFINITION` as follows.

```
receiver[p:receivers]: MODULE =
BEGIN
INPUT
  vecs: ARRAY relays OF vvec,
  pc: stage
LOCAL
  inv: rvec
OUTPUT
  vote: vals
DEFINITION
  inv = [[i:relays] vecs[i][p]]
TRANSITION
[
  ([] (i: vals):
    pc = 3 AND 2*count(inv, i) > n --> vote' = i
   )
[]
  ELSE --> vote' = missing_v
]
END;
```

The final step is to add the receivers in to the `system` specification as follows.

```
system: MODULE =
   controller
     || source
     || (WITH OUTPUT vecs: ARRAY relays OF vvec
         WITH INPUT s_out: rvec
         WITH INPUT rf: ARRAY relays OF faults
           (|| (i:relays): RENAME r_in TO s_out[i],
                                  r_out TO vecs[i],
                                  rf TO rf[i]
                           IN relay[i]))
     || (WITH OUTPUT votes: vvec
           (|| (x:receivers):  RENAME vote TO votes[x]
                               IN receiver[x]));
```

All we need to do now is to specify the properties we wish to examine. However, before we get to the properties of real interest, it will be prudent to check that our specification satisfies some elementary expected properties. The following are simple *liveness* properties that help assure us that our specification makes some progress.[1]

```
live_0: THEOREM system |- F(pc=4);
live_1: THEOREM system |- G(F(pc=5));
live_2: THEOREM system |- F(G(pc=5));
```

The assertion language is not primitive to SAL but is defined by the analyzer employed. Currently, SAL has five analyzers: these are the explicit-state, symbolic, bounded, infinite-bounded, and witness model checkers (the first of these is provided by SAL 1, the other four by SAL 2). The witness model checker uses Computation Tree Logic (CTL) while the others use Linear Temporal Logic (LTL), but with the lexical conventions of CTL (i.e., `G` rather than $\Box$ for *henceforth*, and `F` rather than $\Diamond$ for *eventually*), as their assertion language.[2] LTL formulas are (implicitly) universally quantified over all traces of the system, so that formula `live_0` asserts that in every trace, the program counter eventually gets to 4. Similarly `live_1` says that from any point in any trace, the program counter eventually gets to 5, while `live_2` says that in any trace the program counter eventually gets to 5 and stays there.

We will use the symbolic model checker to examine these properties. The specification developed here is available at `http://www.csl.sri.com/~rushby/specs/om1.`

[1]We will see later that these properties are true for deadlocked systems, and thus provide absolutely no assurance of progress, but they do serve to introduce the syntax.

[2]Actually, all the model checkers accept both CTL and LTL; those whose native assertion language is LTL operate by attempting to translate CTL assertions into LTL, and vice versa for those whose native language is CTL. Since CTL and LTL are incomparable, the translation attempts will sometimes report failure.

sal: if you download this into a file called om1.sal, you will be able to perform the following commands.

Before using the model checker, we should make sure that the specification typechecks.

```
sal-wfc om1
```
Command

This invokes the SAL well-formedness checker, sal-wfc, on the file om1.sal; if the response is anything other than Ok, you will need to understand and correct the error before proceeding.

To check the simple properties, we use commands such as the following, which invokes the symbolic model checker, sal-smc, on property live_0.

```
sal-smc om1 live_0
```
Command

If you would like to see more of what is going on, increase the verbosity level as in the following examination of live_1.

```
sal-smc -v 3 om1 live_1
```
Command

All of these examples should produce the answer Proved in a couple of seconds.

Now we can begin to explore the properties of real interest. The first is validity, which requires that when the algorithm has completed (i.e., pc = 4), and when we have a nonfaulty source (i.e., sf = none), then the vote of every receiver should equal the correct_v. We want this to be true everywhere, so we use the G modality and obtain the following assertion.

```
validity: THEOREM system |-
  G(pc=4 AND sf=none => FORALL(x:receivers):votes[x]=correct_v);
```
Preliminary

Now we model check it.

```
sal-smc -v 3 om1 validity
```
Command

Perhaps to our surprise, this invocation produces the result Invalid, and a counterexample. Examining the last step of the counterexample, we see that two of the relays are manifest faulty, so that the receivers have no majority and choose the value missing_v.

```
Step 3:                                                    Counterexample
--- System Variables (assignments) ---
inv[1][1] = 0;
inv[1][2] = 0;
inv[1][3] = 4;
inv[2][1] = 0;
inv[2][2] = 0;
inv[2][3] = 4;
pc = 4;
rf[1] = manifest;
rf[2] = manifest;
rf[3] = none;
s_out[1] = 4;
s_out[2] = 4;
s_out[3] = 4;
sf = none;
vecs[1][1] = 0;
vecs[1][2] = 0;
vecs[2][1] = 0;
vecs[2][2] = 0;
vecs[3][1] = 4;
vecs[3][2] = 4;
votes[1] = 0;
votes[2] = 0;
```

We obviously need to impose some restrictions on the numbers and kinds of faults that can be present. This suggests we need a function that counts the number of faults present, but we should also weight them in some way. We can leave the weighting parametric by allowing the `fcount` function to take as an argument a `weights` function that maps faults to numbers in the range 0 to 3. In addition to the `weights` function, the function `fcount` is supplied with an array giving the fault status of the `relays`, and the fault status of the `source`. It is defined in terms of a recursive helper function `fcount_h` in the usual way.

```
fc: TYPE = [0..3*(n+1)];

fcount_h(a: ARRAY relays OF faults, acc: all, i: relays,
        weights: [faults -> [0..3]]): fc =
  IF i=1 THEN acc + weights(a[i])
    ELSE fcount_h(a, acc + weights(a[i]), i-1, weights)
  ENDIF;

fcount(a: ARRAY relays OF faults, s: faults,
       weights: [faults ->[0..3]]): fc =
  fcount_h(a, weights(s), n, weights);
```

Since we know that the number of Byzantine faults that can be tolerated is less than a third of the number of nodes, we conjecture that suitable weights will count arbitrary faults

as 3, symmetric as 2, and manifest as 1.[3] This particular mapping is defined below as the function `wts`, and then used in a revised specification of validity that requires `fcount` to be less than the number of nodes.

```
wts(x: faults): [0..3] =
  IF x=arbitrary THEN 3
     ELSIF x=symmetric THEN 2
     ELSIF x=manifest THEN 1
     ELSE 0
   ENDIF;

validity: THEOREM system |-
  G(pc=4 AND sf=none AND fcount(rf, sf, wts) < n =>
    FORALL (x:receivers): votes[x]=correct_v);
```

Perhaps again to our surprise, this invocation produces the result `Invalid` and exactly the same counterexample as last time.

Before we investigate why this is so, let us first examine some model checking issues. On a typical 2GHz machine, the counterexample is found in under 2 seconds. If we change the value of `n` to 4, it takes nearly 3 seconds, and for 5 it takes 31 seconds. Although these times are quite good (the number of reachable states in the `n = 5` case is greater than 7×10^{17} or 700 quadrillion), there are several things we can do to improve them somewhat. First, we can specify `--disable-traceability`: this means that counterexamples are no longer able to indicate which transition fired at each step, but it saves many BDD variables and it reduces the time taken in the `n = 5` case to 12 seconds. Then we can specify the `--backward` search option, and that reduces the time to just over 1 second (`--backward` without `--disable-traceability` takes about 8 seconds).

Command
```
sal-smc -v 3 --disable-traceability --backward om1 validity
```

Whereas disabling traceability will always speed things up, backward search is only sometimes effective (for true properties, it works best for those that are inductive).

For safety properties (i.e., simple `G` properties), *bounded* model checking is an attractive alternative to symbolic model checking when we are expecting to find a counterexample rather than to verify the property. The SAL bounded model checker finds a counterexample for the `n = 5` case in 4 seconds without disabling traceability using the following command (which instructs it to restrict its search to counterexamples of length 3).

Command
```
sal-bmc -v 3 --depth=3 om1 validity
```

[3]This is a deliberately naïve weighting, based on specious reasoning. One of the exercises seeks a correct treatment.

Having seen how to get faster counterexamples, we return to consider why we are getting them. The counterexamples we have seen all have manifest faulty relays but, on reflection, we realize that we would obtain similar counterexamples if we replaced the manifest faults by symmetric ones, and this contradicts our intuition that manifest faults should be easier to deal with than symmetric ones (and are therefore weighted less). Further reflection exposes the problem: the algorithm makes no distinction among fault types and does not "deal with" manifest faults at all, so they are just as potent as symmetric faults. Since manifest faults are, by definition, detectable by all correct receivers, a suitable way to deal with them is to remove `missing_v` values from consideration in the majority vote. This would eliminate our counterexamples, because the `missing_v` values would no longer overwhelm the `correct_v` values. To achieve this, we change the guard in the receiver from

Current
```
([] (i: vals):
    pc = 3 AND
      2*count(inv, i) > n --> vote' = i)
```

to the following.

Improvement
```
([] (i: [1..n+1]):
    pc = 3 AND
      2*count(inv, i) > n - count(inv, missing_v) --> vote' = i)
```

Observe there are two changes here: the multicommand is changed to exclude the `missing_v` case, and the vote calculation requires a majority among only the values *different* to `missing_v`. This modified vote is called the *hybrid* majority; its use changes the overall Oral Messages algorithm to the variant introduced by Thambidurai and Park as "Algorithm Z" [TP88]. Observe that we are using model checking here for *design exploration*: model checking allows us to gain insight into our algorithm and hence to improve it. This use of model checking in the design loop is a valuable adjunct to its better-known uses for debugging and verification [SRSP04].

With this change, the SAL model checker succeeds in verifying the validity property. As before, the time taken to examine the property increases sharply with `n`, unless backward search is used and traceability is disabled (e.g., `n = 5` requires 71 seconds).

Bounded model checking is usually employed only to look for counterexamples, but SAL is also able to use it to perform verification by k-induction [dMRS03]. Since we know the algorithm has three stages, it is natural to use 3-induction, and this succeeds in verifying the property in 12 seconds using the following command.[4]

Command
```
sal-bmc -v 3 --depth=3 --induction om1 validity
```

[4] Since the algorithm takes exactly three stages, it is clear that the inability to find a counterexample by bounded model checking to depth 3 is equivalent to verification. However, this claim depends on our intuitive understanding of the algorithm. Induction at depth three (whose base case requires the absence of a counterexample at this depth) avoids this reliance on intuition.

Symbolic and bounded model checking use completely different methods and underlying technologies (BDDs and SAT solving, respectively), so it is quite often the case that one is much faster than the other on any particular example—and even when not, as here, it is valuable to be able to cross-check their results.

Having gained experience with the `validity` property, it is now quite easy to specify the `agreement` property as follows. This is easily verified by either symbolic or bounded model checking for the original `n = 3` case. Larger cases are left to the exercises.

```
agreement: THEOREM system |-
  G(pc=4 AND fcount(rf, sf, wts) < n =>
    FORALL (x, y:receivers): votes[x]=votes[y]);
```

3 Exercises

The following exercises require changes to the specification and will help develop experience in using the SAL language and its tools. Hints are in the appendix at the back.

3.1 Exploring the Agreement Property

Examine the `agreement` property for increasing values of `n`. You will obtain a counterexample at some point. Examine the counterexample and identify the systematic source of the problem. Modify the specification of the `agreement` property to exclude this case and verify that the property is now true for values up to that `n` (and beyond if your machine is fast enough).

The `agreement` property is more challenging for model checking than `validity`. Examine the growth of the time required to model check the `agreement` property as `n` increases. Explore use of the different options to both the symbolic and bounded model checkers.

3.2 Detecting Flawed Specifications and Properties

Before we draw conclusions from model checking, we need to be sure that the specifications of the system and the properties examined really mean what we think they mean. Try deleting the `G` at the front of the validity property and see what happens. How do you explain it? Restore the `G` and change the antecedent of the property to something obviously false and see what happens. How do you explain it? How can we be sure we avoid these kinds of dangers in real life?

3.3 A Switch Module

It would preferable if the input to each `receiver` module were the `rvec` directed to that receiver. We saw that this is difficult to arrange because the collective output of the `relays`

is an `ARRAY relays OF vvec`. Introduce a `switch` module whose only purpose is to "rotate" this output to an `ARRAY receivers of rvec` so that is becomes possible to use the preferred form of input to the `receivers`. See if you can do this without needing to change the staging of the modules.

3.4 Precise Characterization of Fault Tolerance

The definition given for the function `wts` and its relationship to `n` in the statements of `validity` and `agreement` may not be optimal. Use counterexamples and verifications to help develop intuition and sharper characterizations for the fault tolerance of this algorithm.

3.5 Improving the Algorithm

Modify the specification to represent the algorithm OMH(1) from [LR93] (which distinguishes a missing value from the *report* of a missing value) and/or ZA(1) from [GLR95] (which uses authentication) and develop sharp characterizations for their fault tolerance.

3.6 Link Faults

It underestimates the fault tolerance of the algorithm if a node must be counted as arbitrary faulty when just one of its outgoing lines has a simple "stuck at" fault. Extend the model to include link faults and develop sharp characterizations of the fault tolerance of the algorithm in terms of combinations of link and node faults.

3.7 Liveness Properties

Modify the specification so that it obviously deadlocks. Show that the liveness properties `live_0`, `live_1`, and `live_2` still hold. Why is this? How would you detect deadlock? You will probably need to read the SAL documentation and maybe some papers on temporal logic to answer this.

References

You can obtain papers that have me as an author from http://www.csl.sri.com/~rushby/biblio.html and can find papers by my colleagues via http://fm.csl.sri.com/fmprog.html

References

[BKS02] Günther Bauer, Hermann Kopetz, and Wilfried Steiner. Byzantine fault containment in TTP/C. In *First International Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)*, Vienna, Austria, June 2002. Available from http://www.hurray.isep.ipp.pt/rtlia2002/full_papers/3_rtlia.pdf. 11

[BM91] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991. 16

[BY92] William R. Bevier and William D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4(6A):755–775, 1992. 10

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag. 3

[dMOS01] Leonardo de Moura, Sam Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2001. Revised August 2003. 3

[dMRS03] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In Warren A. Hunt, Jr. and Fabio Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag. 27

[DW78] Daniel Davies and John F. Wakerly. Synchronization and matching in redundant systems. *IEEE Transactions on Computers*, C-27(6):531–539, June 1978. 1

[FTC95] IEEE Computer Society. *Fault Tolerant Computing Symposium 25: Highlights from 25 Years*, Pasadena, CA, June 1995. IEEE Computer Society. 33

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, September 1995. IEEE Computer Society. 29, 34

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003. 11

[LR93] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society. Reprinted in [FTC95, pp. 438–447]. 29, 34

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. 1, 10

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996. 3

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980. 1, 10

[Rus92] John Rushby. Formal verification of an Oral Messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992. Also available as NASA Contractor Report 189704, October 1992. 10

[SRSP04] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The International Conference on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.

[SWR02] Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified Byzantine agreement in presence of link faults. In *The 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 608–616, Vienna, Austria, July 2002. IEEE Computer Society. 35

[TP88] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society. 11, 27

[You97] William D. Young. Comparing verification systems: Interactive Consistency in ACL2. *IEEE Transactions on Software Engineering*, 23(4):214–223, April 1997. 10

A Hints for Exercises

A.1 Exploring the Agreement Property

Take a look at [LR93]. A fix is to disallow manifest faults in the source.

A.2 Detecting Flawed Specifications and Properties

If we omit the `G`, then the assertion applies to just the initial state, where `pc=1`, so the property is vacuously true because its antecedent is false. Similarly, restoring the `G` and making the antecedent false results a property that is true for trivial reasons. Detecting these kinds of problems is called "vacuity detection." Suitable methods are to make sure that the antecedent is true somewhere, but then we have to be careful about vacuity in liveness properties (see a later question).

A.3 A Switch Module

To avoid affecting the staging, define the new values of the switch output in terms of the new values of its inputs.

A.4 Precise Characterization of Fault Tolerance

Check out the formulas and analyses in [LR93] and [GLR95].

A.5 Improving the Algorithm

It is not necessary to model authentication directly; all that is necessary is to eliminate those fault behaviors that authentication would prevent. Consider the modeling of nonces in the SAL treatment of the Needham-Schroeder protocol (available at `http://www.csl.sri.com/users/rushby/abstracts/needham03`).

A.6 Link Faults

Take a look at [SWR02].

A.7 Liveness Properties

You can deadlock the system by removing the `[] ELSE -->` case from, say, the `relay` module. Liveness properties are evaluated only over infinite traces. If there are no infinite traces, the property is vacuously true. Use `sal-deadlock-checker` to check for deadlocks.