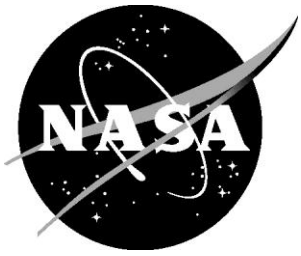
NASA/CR–2013-217960



Modeling and Analysis of Asynchronous Systems Using SAL and Hybrid SAL

Ashish Tiwari and Bruno Dutertre
SRI International, Menlo Park, California

February 2013

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

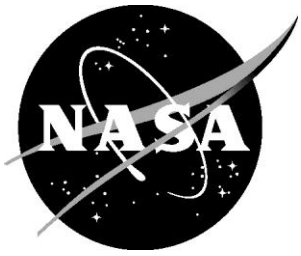
- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2013-217960



Modeling and Analysis of Asynchronous Systems Using SAL and Hybrid SAL

Ashish Twari and Bruno Dutertre
SRI International, Menlo Park, California

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

February 2013

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

We present formal models and results of formal analysis of two different asynchronous systems. We first examine a *mid-value select module* that merges the signals coming from three different sensors that are each asynchronously sampling the same input signal. We then consider the *phase locking protocol* proposed by Daly, Hopkins, and McKenna. This protocol is designed to keep a set of non-faulty (asynchronous) clocks phase locked even in the presence of Byzantine-faulty clocks on the network. All models and verifications have been developed using the SAL model checking tools and the Hybrid SAL abstractor.

Contents

1 Introduction

Flight-critical control systems rely on redundant channels to tolerate failures in hardware and software components. Properly coordinating and managing redundancy in the presence of faults is critical to achieving high reliability. Two general classes of fault-tolerant architectures are commonly used for this purpose:

- *Synchronous architectures* rely on a common time base that enables the redundant channels to work synchronously. Once such a synchronous time base is established, different processors can work as exact deterministic replicas of each other. Fault-tolerant consensus algorithms ensure that all non-faulty replicas perform the same computation on the same input, and thus agree on the control output.

- *Asynchronous architectures* do not attempt to synchronize the different channels. Each processing component is driven by its own local clock. Clocks can be out of phase and drift from each other. As a consequence, the control processors sample sensors and other input at slightly different points in time, which prevents them from reaching exact agreement. The control signals produced by different channels cannot be guaranteed to be equal. Voting and equalization logic is then employed to compensate for the difference between these control signals.

We examine two example systems that belong to each of these two classes of architectures. Both systems are asynchronous and are modeled formally using the SAL and Hybrid SAL tools [2, 5, 6]. The resulting SAL models are available at `http://www.csl.sri.com/users/bruno/vvfcs.html`.

- Our first example is a mid-value select module that merges the signals coming from three different sensors that are each asynchronously sampling the same input signal. This module is part of a larger control system with an asynchronous architecture, It combines signals from three asynchronous components into a single output.

- Our second example is the clock phase locking protocol proposed by Daly, Hopkins, and McKenna [1]. This protocol keeps a set of hardware clocks in phase lock, even if some of the clocks in the system fail. This protocol is one of the first published protocol designed to tolerate components that fail in an arbitrary, possibly asymmetric manner, that is, what we now call *Byzantine failures*.

The goal of the work presented here is to evaluate the possible mechanisms for modeling and analyzing different kinds of asynchronous systems. For the two case studies listed above, we present two different approaches for modeling asynchronous systems:

- In the first approach, asynchronous sampling is modeled as nondeterministic selection from a set of possible sample values; that is, different asynchronously executing components will each nondeterministically select (possibly different) values from a set of values.

- In the second approach, asynchronous effects are captured as synchronous events by performing a time abstraction; that is, events that occur at different, but very close, time instants are fired at the same abstract time, and the abstract time is advanced when no asynchronous event happens.

We provide more details below in the two sections dedicated to each of the two case studies.

2 Robust Asynchronous Control

The asynchronous case study consists of a control system that contains redundant controllers and redundant actuator interface units for tolerating faults. The system is asynchronous. All components, including controllers and actuator interface units, have distinct clocks, which may be out of phase and drift from each other. The system does not include any clock synchronization mechanism.

```
mvs(x1, x2, x3, b1, b2, b3, x) :
   // x1,x2,x3 are the three sensor values,
   // b1,b2,b3 are the corresponding valid bits,
   // x is the previous output value
   if b1 is False then x1 = x endif
   if b2 is False then x2 = x endif
   if b3 is False then x3 = x endif
   if (not b1 and not b2) then
      newx = x3
   else if (not b1 and not b3) then
      newx = x2
   else if (not b2 and not b3) then
      nexx = v1
   else
      newx = (x1 + x2 + x3) - max(x1,x2,x3) - min(x1,x2,x3)
   return newx
```

Figure 1. Pseudocode for the Mid-Value Select Component

The key challenge in analyzing such systems arises from the asynchronous sampling and selection employed by the redundant controllers and actuators. One of the main components of the asynchronous control case study is the mid-value select module that is used to collect and merge values coming from identical, but asynchronously executing, components. In this section, we present the mid-value select algorithm, its formalization in SAL, and then the results of formally analyzing it against a collection of requirements. We note here that we only formalize a small subsystem of the whole asynchronous case study.

2.1 Mid-Value Select

The pseudocode for the mid-value select (MVS) algorithm is given in Figure 1. The algorithm takes three input values, x_1, x_2, x_3 with corresponding validity bits b_1, b_2, b_3, and produces an output $newx$. If bit b_i is true then input x_i is assumed valid. Otherwise, x_i is assumed faulty and it is not taken into account in the computation. The inputs x_1, x_2, x_3 to the mid-value select algorithm are obtained by asynchronously sampling values of some continuous signal. If all three validity bits are true, the mid-value select algorithm returns the middle value of the three inputs. If one b_i is false, then the corresponding x_i is ignored; it is replaced by the mid-value x computed in the previous iteration. If two inputs are invalid, then the remaining good value is forwarded as the output.

We now describe the process of formal modeling the above system. The continuous signal can be the output of some plant model described using ordinary differential equations. The mid-value select component is modeled using a regular (discrete) SAL module. The interesting question that we answer below is how to link the continuous plant and the discrete mid-value select module that takes as input three different asynchronous samples of some output signal produced by the plant.

Figure 2 presents a Hybrid SAL model of the complete system. The final system model is a synchronous composition of three modules called `plant`, `mvs`, and `asynchronousBus`. We assume that the `plant` module outputs a continuous sinusoidal signal y defined by the system of ordinary differential equations

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x. \end{aligned}$$

```
MVS: CONTEXT =
BEGIN
  faulty:  BOOLEAN = FALSE ;
  reallyfaulty:  BOOLEAN = FALSE ;
  dt:  REAL = 0.2 ;
  e:  REAL = 0.05 ;

  midval(y1:  REAL, y2:  REAL, y3:  REAL): REAL = as in Figure 3

  plant:  MODULE =
  BEGIN
    OUTPUT x, y, ylb, yub:  REAL
    INITIALIZATION x = 1; y = 0;
    TRANSITION [ TRUE --> xdot' = y; ydot' = -x ]
  END;

  mvs:  MODULE = as in Figure 3

  asynchronousBus:  MODULE =
  BEGIN
    INPUT ylb, yub:  REAL
    OUTPUT x1, x2, x3:  REAL
    OUTPUT b1, b2, b3:  BOOLEAN
  INITIALIZATION
    x1 = yub; x2 = yub; x3 = yub;
    b1 = TRUE;
    b2 = NOT(reallyfaulty);
    b3 = NOT(faulty)
  TRANSITION
    [ TRUE --> % faulty or not choose x1,x2,x3 correctly
       x1' IN {z:REAL| (ylb' <= z AND z <= yub') OR
                       (ylb' >= z AND z >= yub')};
       x2' IN {z:REAL| (ylb' <= z AND z <= yub') OR
                       (ylb' >= z AND z >= yub')};
       x3' IN {z:REAL| (ylb' <= z AND z <= yub') OR
                       (ylb' >= z AND z >= yub')};
    ]
  END;

  system:  MODULE = asynchronousBus || plant || mvs ;
END
```

Figure 2. Hybrid SAL Model of the Mid-Value Select Component

```
midval(y1:  REAL, y2:  REAL, y3:  REAL): REAL =
  IF y1 <= y2 THEN
     (IF y2 <= y3 THEN y2 ELSIF y1 <= y3 THEN y3 ELSE y1 ENDIF)
  ELSE
     (IF y1 <= y3 THEN y1 ELSIF y2 <= y3 THEN y3 ELSE y2 ENDIF)
  ENDIF;

mvs:  MODULE =
 BEGIN
   INPUT x1, x2, x3:  REAL
   INPUT b1, b2, b3:  BOOLEAN
   OUTPUT x:  REAL
   INITIALIZATION x = 0
   TRANSITION
   [ TRUE -->
       x' = midval(
            IF b1' THEN x1' ELSIF NOT(b2') THEN x3'
            ELSIF NOT(b3') THEN x2' ELSE x ENDIF,
            IF b2' THEN x2' ELSIF NOT(b1') THEN x3'
            ELSIF NOT(b3') THEN x1' ELSE x ENDIF,
            IF b3' THEN x3' ELSIF NOT(b1') THEN x2'
            ELSIF NOT(b2') THEN x1' ELSE x ENDIF)
   ]
 END;
```

Figure 3. The Mid-Value Select Algorithm in SAL

This is enough for illustrating the analysis approach, but a more complex model could be substituted for the `plant` module of Figure 2.

The Hybrid SAL model uses Boolean constants `faulty` and `reallyfaulty` that can be set to true or false to explore different fault scenarios. If both constants are false, then there are no faults in the system. If `faulty` is true, then one sensor value (namely, x_3) is faulty. And if both `faulty` and `reallyfaulty` are true, then the two sensor values x_2 and x_3 are assumed faulty.

The `mvs` module takes three input values x_1 to x_3, and three valid bits b_1 to b_3, and it outputs one value computed according to the algorithm described in Figure 1. The corresponding SAL module is shown in Figure 3 and it uses the `midval` function that is shown in the same figure. Finally, the `asynchronousBus` module takes in the values produced by the `plant` and uses them to compute the inputs for the `mvs`. To model the asynchronous sampling, we model the output from the plant not as a single real-valued signal, but as an interval $[ylb, yub]$. This interval is defined as the set of values that the output y can take in the time interval $[dt - e, dt]$, where dt and e are fixed parameters defined in the SAL context.

Thus, the idea for modeling asynchronous systems consists of modeling the plant so that it generates a set of states (such as an interval) as its output. The `asynchronousBus` then takes the set and nondeterministically selects multiple values from this set. This is our model for asynchronous sampling.

2.2 Discrete Abstraction

In the `plant` module of Figure 2, we have not shown how the `plant` computes `ylb` and `yub`. This is because the expression for `ylb` and `yub` will be derived by the Hybrid SAL relational abstractor automatically from the given (ordinary differential equation) dynamics of the variables x and y, and the time constants dt and e. Our current version of the Hybrid SAL relational abstractor does not yet support this feature. We

plan to add this feature to the Hybrid SAL relational abstractor in the next few months.

On this specific example, we intend the abstractor to produce an abstract discrete model similar to the one shown in Figure 4. This discrete model was constructed by hand, using the relational-abstraction technique described in [6], adjusted to compute constraints on the variables `ylb` and `yub` rather than on a single value y. The constraints on `ylb` and `yub` are derived by letting variables x and y evolve for a time interval of length dt (from their current values) according to the differential equations, and recording the value of y at both the end of this interval (i.e., at time dt) and at the intermediate point $dt - e$. The abstraction introduces an auxiliary Boolean variable `inc` that indicates whether y is increasing or decreasing at the beginning of the interval $[0, dt]$.

In addition to the abstracted plant, Figure 4 contains an observer module called `deadzoneMonitor`. This module detects *deadzones*, that is, situations during which the input changes, but the output does not. If this behavior is observed, then the `deadzoneMonitor` module sets the `flag` variable to `true`. Composing this observer with the `system` module lets us check for deadzone behavior in the model.

Every state transition of the full `system_monitor` of Figure 4 consists of the following three steps:

- First, the `plant` module updates its outputs. The `plant` module computes the lower and upper-bound for the variable y reached in the time interval $[dt - e, dt]$.
- The `asynchronousBus` module then reads in the new lower- and upper-bounds for y and non-deterministically chooses three values from that range. This models asynchronous sampling. The three values are output by the `asynchronousBus` module. Note the `asynchronousBus` module reacts to the primed forms of the inputs, indicating that it uses the updated values for `ylb` and `yub` (generated in the first step).
- Finally, the `mvs` module applies the mid-value select computation to the three values it gets the `asynchronousBus`. The result is the output variable `x`.

This shows how we model asynchronous sampling using SAL synchronous composition of base modules.

2.3 Properties and Analysis

The SAL model of the asynchronous system is an infinite-state model and it is not possible to verify it using exhaustive state space exploration. It can only be analyzed using the SAL bounded model checker for infinite systems (`sal-inf-bmc`). All proofs use `sal-inf-bmc` as a k-induction prover (i.e.. by giving option `-i`) to `sal-inf-bmc`).

In Figure 5, we have briefly enumerated some properties that can be formally checked against the models `system` and `system_monitor`. Property `p0` states that the output of the mid-value select is not too far off from the actual value of the signal. This property can be verified for all fault scenarios. It is proved by invoking `sal-inf-bmc` as follows

```
sal-inf-bmc -i -d 1 MVS.sal p0
```

This calls `sal-inf-bmc` to check property `p0` of a SAL context contained in file `MVS.sal` using k-induction. For this property, induction depth of 1 is sufficient (i.e., k=1). This is specified by giving option `-d 1` on the command line.

Properties `l1, l2, l3` are listed just because they are sometimes useful as auxiliary lemmas for proving other properties by k-induction. Lemma `l3` is provable in all fault scenarios in the same way as property `p0` (i.e., by induction at depth 1). Lemma l1 is provable by the same method if there are no faults. Lemma l2 is provable if there is a single fault. Property `p2` is false and is included just to debug the model.

```
MVS: CONTEXT =
BEGIN
  global parameter declarations as in Figure 2
  midval(y1:  REAL, y2:  REAL, y3:  REAL): REAL = as in Figure 2

  plant:  MODULE =
  BEGIN
    OUTPUT yub, ylb:  REAL
    LOCAL inc:  BOOLEAN
  INITIALIZATION
    yub = 0; ylb = 0; inc = TRUE;
  TRANSITION
    [ yub>= -0.8 AND yub <= 0.8 AND inc -->
         yub' = yub + dt * 1 ; ylb' = yub + (dt - e) * 1;
    []
      yub >= -0.8 AND yub <= 0.8 AND NOT(inc) -->
         yub' = yub + dt * (-1) ; ylb' = yub + (dt - e) * (-1);
    []
      ((yub <= -0.8 AND yub >= -1) OR yub >= 0.8) AND NOT(inc) -->
         yub' = yub + dt * (-0.1) ; ylb' = yub + (dt - e) * (-0.1);
         inc' = IF yub' <= -1 THEN TRUE ELSE FALSE ENDIF ;
    []
      ((yub >= 0.8 AND yub <= 1) OR yub <= -0.8) AND inc -->
         yub' = yub + dt * (0.1) ; ylb' = yub + (dt - e) * (0.1) ;
         inc' = IF yub' >= 1 THEN FALSE ELSE TRUE ENDIF ;
    ]
  END;

  mvs:  MODULE = as in Figure 2
  asynchronousBus:  MODULE = as in Figure 2
  system:  MODULE = asynchronousBus || plant || mvs ;

  deadzoneMonitor:  MODULE =
  BEGIN
    INPUT x, yub:  REAL
    OUTPUT flag:  BOOLEAN
  INITIALIZATION flag = FALSE
  TRANSITION
    [ NOT (yub = yub') --> flag' = (x = x')
    [] ELSE -->
    ]
  END;

  system_monitor:  MODULE = system || deadzoneMonitor ;
```

Figure 4. Abstracted SAL Model of the Mid-Value Select Component with Observer

```
p0:  THEOREM
  system_monitor |- G((x >= yub AND x - yub <= 0.05) OR
                      (x <= yub AND yub - x <= 0.05));

l1:  THEOREM system_monitor |- G ( b1 AND b2 AND b3 );

l2:  THEOREM system_monitor |- G ( b1 AND b2 AND NOT(b3) );

l3:  THEOREM system_monitor |- G (-1.02 <= yub AND yub <= 1.02);

p1:  THEOREM system_monitor |- G ( flag = FALSE ) ;

p2:  THEOREM system_monitor |- G ( inc ) ;
```

Figure 5. Correctness Properties for the Mid-Value Select Component

Property `p1` says that `flag` is never `true`, which means that there is no instance of deadzone behavior. Changing the value of parameters `dt` and `e` influences the validity of this property. When `e` is small, then there is no deadzone behavior since a small value of `e` indicates more "synchronization" between the three elements. When `e` is large, then deadzones can be observed even in the absence of faults. Perhaps surprisingly, the behavior of this model is not affected much by the presence or absence of faults. For example, if `e` is set to 0.05 and `dt` is `0.2` then no deadzone is observed even if one or two sensors are faulty. In all fault scenarios, property `p1` is provable by induction at depth $k = 2$. On the other hand, for the same value of `d`, if `e` is increased to `0.1`, then deadzone can be observed even in the fault-free case.

3 Draper Phase Locking Protocol

We now discuss our SAL model of the fault-tolerant clock system proposed by William M. Daly, Albert L. Hopkins, Jr., and John F. McKenna from MIT Draper Labs [1]. This system consists of an array of identical clock elements that communicate with each other to produce a number of phase-locked signals. If less than a third of these clock elements are faulty, then all non-faulty outputs are in phase lock. This property holds even if the faulty components are asymmetric, that is, even in the presence of Byzantine faults.

We have developed several SAL models of this system using the same modeling approach. A previous model (available at `http://www.csl.sri.com/users/bruno/vvfcs.html`) encoded the rules of the protocol at an abstract level. Using this model, we concluded that the protocol can *maintain* clocks in phase locks even in the presence of Byzantine faults. In other words, if the clocks are initially in phase lock, they will remain phase locked, even in the presence of Byzantine fault. However, in this model, the algorithm did not ensure that phase lock is reached from arbitrary initial configurations. There were scenarios in which a Byzantine clock caused the good clocks to form two disjoint cliques running in opposite phase.

We have refined this initial SAL model by modeling more precisely the hardware circuit of the clock elements presented in the paper by Daly et al. [1]. Using this refined model, we can prove that good clocks do eventually synchronize, from any initial configuration, under the assumption that the faulty clock does not *change* more than once at a given time instant. i.e., it is not allowed to change too fast. In other words, we need to add a test in each clock element that checks to make sure that other clock elements do not send it different values too frequently.

We now describe the modeling and analysis results in more detail.

3.1 Draper Clock Phase Locking

The Draper clocking system consists of n identical clock elements that communicate with each other to produce a number of phase-locked clock signals. If $n \geq 3f + 1$, then the clocking system can tolerate f Byzantine faults and maintain the $2f + 1$ good clocks in phase-locked state.

Since the clock elements execute asynchronously, the phase-locked condition is defined as follows: two clocks are phase-locked to within t_s time units if state changes in the two clocks occur within t_s time units of each other. The parameter t_s is a function of the communication delay between the clock elements and is assumed to be significantly smaller than the time period of the clocks.

Each individual clock element receives the state of the other clocks. It uses this information to compute various quorum functions on the clock states. Based on the values of the computed quorum functions, the clock element possibly triggers one-shot monostable multivibrators, which then set the value of the output clock via two flip flops.

In this asynchronous system, state changes happen at two different time scales:

- A fast time scale where multiple state changes happen in a relatively short time span (microsteps).
- A slow time scale where no state change occurs until some timer expires (macrostep), which then initiates a cascade of microsteps.

3.2 Formalization in SAL

We formalize the Draper clocking system by first abstracting time. Each clock element in the original system has two one-shot multivibrators, called z10 and z11, with time period of 860 ns and 200 ns respectively. It also has two flip-flops, z12a and z12b, where the value stored in flip-flop `z12a` is the current state of the clock. Thus, the state of a clock element consists of

(a) the Boolean values stored in the two flip-flops and

```
N: NATURAL = 3;   % number of NON-FAULTY nodes
f:  NATURAL = 1;   % number of faulty nodes
Node:  TYPE = [1 ..  N];
tMax:  NATURAL = 11;  % time period of z10
tMin:  NATURAL = 3;   % time period of z11
Time:  TYPE = [-1 ..  tMax];

min(x,y:  ARRAY [1..N] OF Time, i:[0..N], a:Time):  Time =
 IF i = N AND a = tMax THEN 0
 ELSIF i = N THEN a
 ELSIF (x[i+1] < y[i+1] OR y[i+1] = -1) AND x[i+1] < a AND x[i+1] > 0
   THEN min(x, y, i+1, x[i+1])
 ELSIF (x[i+1] >= y[i+1] OR x[i+1] = -1) AND y[i+1] < a AND y[i+1] > 0
   THEN min(x, y, i+1, y[i+1])
 ELSE min(x, y, i+1, a) ENDIF;

sum(x:  ARRAY [1..N] OF BOOLEAN, i:  [0..N], a:  [0..N]): [0..N] =
 IF i = N THEN a ELSIF x[i+1] THEN sum(x, i+1, a+1)
 ELSE sum(x, i+1, a) ENDIF;
```

Figure 6. Constants and functions used in the Draper Clock Phase Locking SAL Model

(b) the time remaining before each one-shot will return to its stable state (i.e. before it will output a falling edge).

If z10 or z11 are in their stable state (i.e., they have not been triggered recently), then we set the time remaining to -1. Since the time periods are 860 ns and 200 ns, hence, at any instant, the time remaining before z10 and z11 fire will either be a value between 0 and 860 ns, or it will be -1.

In the formal model of a clock element, we replace the continuous time range by the integer interval $[-1 \ldots tMax]$. In our experiments, $tMax$ was fixed to 11 time units and the time periods of the two one-shots was fixed to 11 and 3 respectively. As seen in Figure 7, the state of each clock element in our model consists of:

- the state of the clock, `c0`, which is a Boolean-valued variable (stored in flip-flop `z12a`)
- the Boolean value stored in the flip-flop `z12b`
- the time remaining in the two one-shots, `z10` and `z11`, which are values in the integer interval $[-1 \ldots tMax]$

The SAL model is parameterized by N, the number of good clocks, and f, the number of faulty clocks. Each clock element receives as input the values of two quorum functions `qni` and `qfi`, which are computed based on the values of all the clock elements. We denote by $Q(b, a)$ the quorum function that has value 1 if a or more of its b inputs are 1, and has value 0 otherwise. Then, `qfi` is the quorum function $Q(N+f, f+1)$ applied to the $N+f$ clocks in the system, and `qni` is the quorum function $Q(N+f, N)$ of the same $N+f$ clocks. In other words, `qfi` is 1 if at least $f+1$ of the clocks are 1 and `qni` is 1 if at least N clocks are 1.

Each clock element also receives another input, namely `timeAdvance`, which is a value in the range $[0 \ldots tMax]$, that indicates how much time has elapsed since the last state transition.

The Draper clocking system model is obtained by composing the clock element models. There is one other module, called `calculator`, that computes the quorum functions and the value of the `timeAdvance`

```
element[i:  Node]:  MODULE =
BEGIN
  OUTPUT z10i, z11i:  Time
  LOCAL z12b:  BOOLEAN
  OUTPUT c0:  BOOLEAN
  INPUT qni, qfi:  BOOLEAN
  INPUT timeAdvance:  Time
  INITIALIZATION
    z10i = tMax - i;
    z11i = -1;
    z12b = TRUE
  TRANSITION
  [
  (NOT(qni) AND qni' AND NOT(qfi) AND qfi') OR
  (qfi AND NOT(qfi') AND qni AND NOT(qni')) -->
    z10i' = tMax-1 ;
    z11i' = IF z11i = -1 THEN tMin ELSE z11i ENDIF ;
    c0' = IF z11i = -1 THEN z12b ELSE c0 ENDIF
  []
  ((NOT(qni) AND qni') OR (qfi AND NOT(qfi'))) AND
  NOT( (qni AND NOT(qni')) OR (NOT(qfi) AND qfi')) -->
    z10i' = tMax-1
  []
  NOT((NOT(qni) AND qni') OR (qfi AND NOT(qfi'))) AND
  ( (qni AND NOT(qni')) OR (NOT(qfi) AND qfi')) -->
    z11i' = IF z11i = -1 THEN tMin ELSE z11i ENDIF;
    z11i' = IF z11i = -1 THEN tMin ELSE z11i ENDIF;
    c0' = IF z11i = -1 THEN z12b ELSE c0 ENDIF
  []
   See remaining transitions in Figure 8
```

Figure 7. Model of an Individual Clock Element in SAL (Part 1)

```
element[i:  Node]:  MODULE =
BEGIN
  ...
  TRANSITION
  continuing from Figure 7
  []
  qni = qni' AND qfi = qfi' AND timeAdvance' > 0 AND
  z10i = timeAdvance' AND NOT(z11i = timeAdvance') -->
    z10i' = tMax-1;
    z11i' = IF z11i = -1 THEN tMin ELSE z11i - timeAdvance' ENDIF;
    c0' = IF z11i = -1 THEN z12b ELSE c0 ENDIF
  []
  qni = qni' AND qfi = qfi' AND timeAdvance' > 0 AND
  z10i = timeAdvance' AND z11i = timeAdvance' -->
    z10i' = tMax-1 ;
    z11i' = -1 ;
    z12b' = NOT(qni) ;
  []
  qni = qni' AND qfi = qfi' AND timeAdvance' > 0 AND
  z11i = timeAdvance' AND NOT(z10i = timeAdvance') -->
    z10i' = IF z10i = -1 THEN -1 ELSE z10i - timeAdvance' ENDIF;
    z11i' = -1 ;
    z12b' = NOT(qni) ;
  []
  qni = qni' AND qfi = qfi' AND timeAdvance' > 0 AND
  NOT(z11i = timeAdvance') AND NOT(z10i = timeAdvance') -->
    z10i' = IF z10i > 0 THEN z10i - timeAdvance' ELSE -1 ENDIF;
    z11i' = IF z11i > 0 THEN z11i - timeAdvance' ELSE -1 ENDIF;
  []
  qni = qni' AND qfi = qfi' AND timeAdvance' = 0 -->
  ]
END;
```

Figure 8. Model of an Individual Clock Element in SAL (Part 2)

```
draperClockv8:  CONTEXT =
BEGIN
  constant and function declarations from Figure 6
  calculator:  MODULE =
  BEGIN
    INPUT c:  ARRAY [1 ..  N] OF BOOLEAN
    INPUT z10, z11:  ARRAY [1 ..  N] OF Time
    OUTPUT qnmf, qfp1:  ARRAY [1 ..  N] of BOOLEAN
    OUTPUT timeAdvance:  Time
    LOCAL smin:  [1 ..  N]
    LOCAL faultc:  ARRAY Node OF [0 ..  1]
    INITIALIZATION
      faultc = [ [i:Node] 0];
      qnmf = [ [i:[1..N]] FALSE ];
      qfp1 = [ [i:[1..N]] FALSE ];
      timeAdvance = 0
    DEFINITION smin = sum(c, 0, 0) ;
    TRANSITION
    [
    timeAdvance <= 0 -->
      qnmf' = [ [i:[1..N]] smin+faultc[i] >= N ] ;
      qfp1' = [ [i:[1..N]] smin+faultc[i] >= f+1 ] ;
      timeAdvance' = IF FORALL (i:[1..N]): (qnmf[i] = qnmf'[i] AND
                                            qfp1[i] = qfp1'[i])
                     THEN min(z10, z11, 0, tMax) ELSE 0 ENDIF
    []
    timeAdvance > 0 -->
      faultc' IN {a:  ARRAY Node OF [0..1] | TRUE };
      qnmf' = [ [i:[1..N]] smin+faultc'[i] >= N ] ;
      qfp1' = [ [i:[1..N]] smin+faultc'[i] >= f+1 ] ;
      timeAdvance' = IF FORALL (i:[1..N]): (qnmf[i] = qnmf'[i] AND
                                            qfp1[i] = qfp1'[i])
                     THEN min(z10, z11, 0, tMax) ELSE 0 ENDIF
    ]
  END;

  element[i:  Node]:  MODULE = from Figure 7

   continued in Figure 10
```

Figure 9. Draper Clock Phase Locking in SAL (Part 1)

```
draperClockv8:  CONTEXT =
BEGIN
  first part of model in Figure 9
  system:  MODULE = calculator ||
    ( WITH OUTPUT c:  ARRAY Node OF BOOLEAN
      WITH INPUT qnmf:  ARRAY Node OF BOOLEAN
      WITH INPUT qfp1:  ARRAY Node OF BOOLEAN
      WITH OUTPUT z10:  ARRAY Node OF Time
      WITH OUTPUT z11:  ARRAY Node OF Time
      (|| (i:  Node):  RENAME c0 TO c[i], qni TO qnmf[i],
                         qfi TO qfp1[i], z10i TO z10[i],
                         z11i TO z11[i]
                         IN element[i] ) ) ;

  monitor:  MODULE =
  BEGIN
    INPUT c:  ARRAY Node OF BOOLEAN
    INPUT timeAdvance:  Time
    OUTPUT flag:  BOOLEAN
    INITIALIZATION flag = FALSE
    TRANSITION
    [
    timeAdvance' > 0 AND FORALL (i:Node):  (c[i] = c[1]) -->
      flag' = TRUE
    []
    timeAdvance' > 0 AND EXISTS (i:Node):  NOT(c[i] = c[1]) -->
      flag' = FALSE
    []
    ELSE -->
    ]
  END;

  system_monitor:  MODULE = system || monitor ;

  See properties in Figure 11
END
```

Figure 10. Draper Clock Phase Locking in SAL (Part 2)

variable. Thus, the `calculator` produces the inputs needed by each clock element. The `calculator` module receives the values of all (non-faulty) clock elements as input and the time remaining in the one-shots of all the clock elements. A key feature of this model is that we do not have a module instance for the faulty clock. We assume exactly one faulty clock in the SAL model, but the model can be easily extended so that it has f faulty clocks.

The `calculator` models the Byzantine nature of the faulty clock — each of the good clock elements can see a different value for the faulty clock. The `calculator` module works as follows:

- It first computes the values of the quorum function for each of the non-faulty clock elements.
- If there is leading or falling edge on any of the quorum values, `calculator` sets `timeAdvance` to zero. Otherwise, it sets `timeAdvance` to the minimum time remaining on any of the one-shots. The `min` function in Figure 6 is used for this purpose. Thus, the `calculator` ensures that all microsteps are "completed" before the next "macrostep" is initiated.
- The Byzantine-faulty clock is allowed to non-deterministically pick the N clock values it wishes to show to the N non-faulty clocks, but only when `timeAdvance` is positive. This essentially disallows the Byzantine faulty clock from changing its value indiscriminately – and an arbitrary number of times – in the same abstract time instant.

This last point is crucial for proving correctness of the Draper clocking system.

3.3 Modeling Decisions

In the actual Draper clocking system, when one clock changes state, all the other clocks will asynchronously observe this event. They may not all see the change at the same time instant, but they will definitely see it within a bounded amount of time. Modeling this behavior exactly in SAL is possible, but leads to very complex formal models — one clock changes state and some clock does not "observe" the change in the same SAL transition. In order to guarantee that every clock *does* observe all such clock change events, one would have to maintain a stack of pending events.

In our formalization, we model events that occur within some bounded time of each other (microsteps) occurring in the same state transition. Thus, in our model, when a clock changes state, all the clocks observe this change and respond to it. This way we do not have to store a stack of pending events. As a result, we are potentially not capturing all possible behaviors of the concrete system. However, we believe that the we are not missing too many behaviors of the concrete system, since the order in which events are processed by different clock elements (order of the microsteps) does not seem to matter for determining the final state of the system.

Finally, the paper by Daly et al. [1] does not provide complete details of the clocking circuit and some aspects are ambiguous. For example, it does not fully specify the behavior of the one-shot multivibrators and the flip-flops. In such cases of ambiguity, we made certain assumptions while building the formal model.

3.4 Verification results

Figure 11 lists some properties of the clocking system. Of particular interest is property `p9` which states that *eventually the clocks are synchronized*. For the above SAL model, the SAL symbolic model checker, `sal-smc`, was able to analyze all the properties. All the properties listed are true, except `p3` and `p4` (which were used to debug the model).

An important observation coming from our verification effort was that it is important to limit the Byzantine faulty clock so that it can not send different clock values to the same clock very frequently. Without this assumption, it is not possible to prove that the system will always eventually reach a state of phase locking.

```
p1:  LEMMA
  system |- F( G ( (FORALL (i:[1..N]): c[i] = c[1])) );

p2:  LEMMA
  system |- F ( ((FORALL (i:[1..N]): c[i] = c[1])) );

p3:  LEMMA
  system |- G ( timeAdvance = 0 );

p4:  LEMMA
  system |- G( EXISTS (i:[1..N]): z11[i] < tMin );

p5:  LEMMA
  system |- G( EXISTS (i:[1..N]): z10[i] > tMin );

p6:  LEMMA
  system |- G( EXISTS (i:[1..N]):
                NOT(z10[i] = -1) OR NOT(z11[i] = -1));

p7:  LEMMA
  system |- F ( ((FORALL (i:[1..N]): c[i] = c[1])) );

p8:  LEMMA
  system_monitor |- F(G ( flag ));

p9:  LEMMA
  system |- F ( G ( ((FORALL (i:[1..N]): c[i] = c[1])) ) );

p10:  LEMMA
  system |- G(F(timeAdvance > 0));

p11:  LEMMA
  system |- G(F(EXISTS (i:[1..N]):
                  (NOT(z10[i] = -1) OR NOT(z11[i] = -1))));

p12:  LEMMA
  system |- G ((FORALL (i:[1..N]): c[i] = c[1]) =>
               G ( (FORALL (i:[1..N]): c[i] = c[1])) );
```

Figure 11. Correctness Properties for the SAL model of the Draper Clock Phase Locking System

Another option would be to not limit the Byzantine faulty clock, but include an *extra check* in each clock element that ignores inputs from a clock if they arrive too frequently. Thus, each clock element is required to perform some form of faulty clock detection. We believe it is not possible to achieve clock synchronization using purely quorum sensing and without any form of fault detection/isolation. Such monitoring mechanisms are employed in the self-stabilizing Byzantine-fault-tolerant clock synchronization protocols proposed by Malekpour [3, 4]. Note that the clocking system described by Daly et al. [1] does not include any provision for such fault isolation.

4 Conclusion

We have presented two possible methods for modeling and verifying asynchronous fault-tolerant systems using the SAL tool chains.

Our first case study is an asynchronous mid-value select module, which is part of a larger fault-tolerant, asynchronous system. In such systems, different functional modules are driven by local clocks that are not synchronized. They sample input at slightly different points in time, thus possibly producing different output, which have to be combined and reconciled in some way to drive actuators. The mid-value select module we have described is a common approach for combining three asynchronously produced input into a single output, while providing support for fault tolerance. For analyzing such a module in a larger context (that may involve continuous dynamics), we aim to represent the mid-value select mechanism in Hybrid SAL. We have proposed an abstraction approach that approximates asynchronous sampling by the non-deterministic selection of state values in an interval. This approach can be automated by modifying the timed-relational abstraction algorithm implemented by Hybrid SAL, so that it considers time intervals rather than time points.

Our second case study is the Byzantine fault-tolerant clock system presented in [1]. We modeled this system by using a discrete-time approximation. The resulting model is a standard SAL model (with no continuous dynamics), that has a finite state space. As a result, the system can be automatically verified using a classical BDD-based model checker. One key property discovered in this case study was the need to restrict the fault model so that the value of Byzantine clocks do not change too fast. Under this assumption, we have proved all key properties of the protocol: phase-locked synchronization is eventually reached from an arbitrary initial configuration of the clocks, and the non-faulty clocks remain in phase lock from then on.

References

1. William M Daly, Alfred L Jr. Hopkins, and John F. McKenna. A Fault-Tolerant Digital Clocking System. In *Proceedings of the 3rd IEEE International Symposium on Fault-Tolerant Computinfg (FTCS 3)*, pages 17–22, 1973. Reprinted in Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years.

2. Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer-Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer-Verlag, July 2004.

3. Mahyar R. Malekpour. A Byzantine Fault-Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems. In *8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'2006)*, volume 4280 of *Lecture Notes in Computer Science*, pages 411–427. Springer, 2006.

4. Mahyar R. Malekpour. A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol. Technical Report NASA/TM-2009-215758, NASA, June 2009.

5. Sriram Sankaranarayanan and Ashish Tiwari. Relationa Abstractions for Continuous and Hybrid Systems. In *Computer-Aided Verification (CAV 2011)*, volume 6806 of *Lecture Notes in Computer Science*, pages 686–702. Springer-Verlag, July 2011.

6. Ashish Tiwari. The HybridSAL Relational Abstractor. In *Computer-Aided Verification (CAR 2012)*, 2012. To appear. Preprint available at `http://www.csl.sri.com/users/tiwari/relational-abstraction/`.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 02 - 2013	Contractor Report	12/2011 - 05/2012

4. TITLE AND SUBTITLE

Modeling and Analysis of Asynchronous Systems Using SAL and Hybrid SAL

5a. CONTRACT NUMBER

NNL10AB32T

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Tiwari, Ashish; Dutertre, Bruno

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

534723.02.02.07.30

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, Virginia 23681

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/CR-2013-217960

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

This report was prepared by SRI International under NASA contract NNL10AB32T with Honeywell International, Inc., Golden Valley, MN.
Langley Technical Monitor: Paul S. Miner

14. ABSTRACT

We present formal models and results of formal analysis of two different asynchronous systems. We first examine a mid-value select module that merges the signals coming from three different sensors that are each asynchronously sampling the same input signal. We then consider the phase locking protocol proposed by Daly, Hopkins, and McKenna. This protocol is designed to keep a set of non-faulty (asynchronous) clocks phase locked even in the presence of Byzantine-faulty clocks on the network. All models and verifications have been developed using the SAL model checking tools and the Hybrid SAL abstractor.

15. SUBJECT TERMS

Asynchronous systems; Clock synchronization; Fault tolerance; Hybrid systems; Model checking

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	24	(443) 757-5802