NASA/CR–2013-217984

Investigating Actuation Force Fight with Asynchronous and Synchronous Redundancy Management Techniques

Brendan Hall, Kevin Driscoll, and Kevin Schweiker
Honeywell International, Inc., Golden Valley, Minnesota

Bruno Dutertre
SRI International, Menlo Park, California

April 2013

NASA/CR–2013-217984

Investigating Actuation Force Fight with Asynchronous and Synchronous Redundancy Management Techniques

Brendan Hall, Kevin Driscoll, and Kevin Schweiker
Honeywell International, Inc., Golden Valley, Minnesota

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

April 2013

Available from:

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

Abstract

Within distributed fault-tolerant systems the term force-fight is colloquially used to describe the level of command disagreement present at redundant actuation interfaces. This report details an investigation of force-fight using three distributed system case-study architectures. Each case study architecture is abstracted and formally modeled using the Symbolic Analysis Laboratory (SAL) tool chain from the Stanford Research Institute (SRI). We use the formal SAL models to produce k-induction based proofs of a bounded actuation agreement property. We also present a mathematically derived bound of redundant actuation agreement for sine-wave stimulus. The report documents our experiences and lessons learned developing the formal models and the associated proofs.

Contents

1 Introduction 5

1.1 Scope 5

1.1.1 Background and Motivation 5

1.2 Domain-Specific Architecture Evolution 6

2 Case-study Architecture Review 7

2.1 Asynchronous Tripplex High Integrity Control 7

2.2 Asynchronous BRAIN-based Ethernet Architecture 9

2.3 Synchronous Two-tier Network Architecture 11

3 A Discussion of the Case-study Architectures 12

3.1 Overview 12

3.2 Assumed Failure Modes 12

4 Formally Modeling Actuation Agreement Using SAL 14

4.1 Formal Model Description 14

4.1.1 Asynchronous Interaction 15

4.1.2 Modeling Synchronous Interaction 16

4.1.3 Fault-Injection 16

4.1.4 MVS Evaluation 17

4.1.5 The Agreement Monitor 18

4.1.6 Model Composition 18

4.1.7 Proving Agreement Properties 19

4.1.8 Additional Model Validation Experiments 19

4.2 Discussion of Initial Model and Findings 20

4.3 Initial Model Checking Performance and findings 21

4.4 An Alternative Timeout Automata Based Abstraction 22

4.4.1 Clock Module 22

4.4.2 Source Module 22

4.4.3 FCM Module 23

4.4.4 MVS Module 23

4.4.5 Fault Injection 24

4.4.6 System Composition 24

4.4.7 Investigating and Proving the System Agreement Properties 24

4.5 Investigating Faults with the Timeout Automata Based Abstraction 26

4.5.1 Proving Agreement with Faults 27
4.6 Run Scripts and Model Source Files 28
5 Mathematical Analysis of Mid-Value-Selection 28
5.1 Inconsistent Omission Error Force Fight 31
6 Conclusions and Future Work 33
References 34

List of Figures

Figure 1 Asynchronous Three Channel Switched Ethernet Architecture 7
Figure 2 Ethernet Brain Based Case-study Architecture 9
Figure 3 Two-tier Synchronous Network Case -study Architecture.. 11
Figure 4 Actuation Force-Fight Instrumentation 12
Figure 5 Formal Model High Level Structure 14
Figure 6 CM Sinusoidal Outputs 28
Figure 8 Disagreement between CM channels and Mid-Value Select.. 30
Figure 7 Mid-value selection from 3CMs, Channel 3 fails at time=3.0 sec. 30
Figure 9 Dual ACE Force Fight - Inconsistent Omission Error.. 31
Figure 10 Force Fight - Inconsistent Omission Error at t=3.0 sec. 31
Figure 11 Triangle Wave used in SAL Analysis. 32
Figure 12 Force Fight from Triangle CM Commands - Single Channel Inconsistent Omissions. 32

1 Introduction

The document has been generated under NASA Task Order NNL10AB32T. It presents the modeling and exploration of the control system case-study architectures presented in [1].

In this document we have constructed formal model abstractions of key system strategies related to redundancy management. We use the models to prove characteristics of case-study architectures. In this initial work these proofs a single property is selected for formal examination. This selected property, colloquially termed *force-fight* denotes the level of command disagreement that exists across redundant actuation interfaces.

1.1 Scope

This work is based on the Phase 2 control system case-studies that are documented in [1]. Although the case studies embody control-system models, the focus of our work is not related to control theory. The focus of this work is the formal investigation of distributed-system redundancy management logic. The control element of the problem is included here only to enable the interaction of the distribution and replication management policies with the higher level requirements of the external control system. For this initial work, the behavior of the control law was abstracted out of the formal representation, to enable simpler bounds of agreement to be formally established.

Full listings of the SAL and Matlab models presented herein are available at the NASA DASHlink site AFCS – Distributed Systems (https://c3.nasa.gov/dashlink/projects/79/).

1.1.1 Background and Motivation

During Phase 1 of this research, most of the system modeling and analysis activities were focused on modeling system communication infrastructures and their associated protocols. However, during the review of the asynchronous case study [2], we learned that many real-world systems neither built upon nor leveraged the layered fault-tolerant services prescribed by formal fault-tolerance theory. In place of structured, layered, fault-tolerant services, these systems implement application-specific, fault mitigation strategies derived from field-proven domain experience. In such systems, as illustrated in [1], the system fault-tolerant strategies are often dispersed throughout the system control-law implementation. This dispersal complicates incremental verification, as the system fault tolerance is coupled with the application's behavior. Consequently, formal validation (i.e., formally proving the correctness and sufficiency of the system fault-tolerance) is also non-trivial. However, given the wide-spread proliferation of these techniques, we believe that developing a formal framework that enables the validation of such system strategies will be very beneficial.

We hope that this analysis will yield more systematic review, and potentially automation, of some of the validation activities required for this class of systems. To this end, as part of the Year 3 efforts, we intend to explore the feasibility of test generation from the system formal model to support the current manually-generated design validation activities. This work supports improved completeness claims with respect to system-level validation activities.

In addition, given that many aspects of this class of system design are based upon years of domain-centric experience, we hope that formally capturing the knowledge associated with these systems will offset the risks associated with retaining this critical knowledge as the current workforce retires.

Finally, by contrasting the performance of the different case-study architectures we further hope to develop some insights about the potential and strengths and weaknesses related to the theoretical fault-tolerance strategies and industrial pragmatic fault-tolerance approaches.

1.2 Domain-Specific Architecture Evolution

Our discussion above explained that the design philosophy of many real-world approaches to fault tolerance has evolved pragmatically as the systems have taken on increasing levels of authority and system responsibility. Osder [3] describes this evolution within the flight control system architectures as analog and later digital electronic technologies where introduced. As these system architectures evolved, the increasing dependence and specific failure models of digital hardware need to be mitigated and domain-specific architectural techniques [4] developed. These techniques were largely influenced by the voting and fault detection strategies used to select among multiple lanes of redundancy. Some early designs leveraged global synchronization to reduce the complexity of cross-lane voters. With system-level synchronization, the error tolerance of the voters can be easily calculated from the system's precision performance. However, the potential brittleness and common-mode influence of the system synchronization service led others to develop asynchronous cross-channel voting strategies [4]. Further background descriptions of synchronous and asynchronous system architectures are given in [5][6].

For commercial flight control, the asynchronous design strategy is most prevalent today. This strategy is an interesting choice, given the complexities of designing and validating such systems, which are complicated due to the inexact agreement across redundant lanes[1]. However, it appears that modern flight control systems do not to require exact agreement. Therefore, the approximate agreement properties possible with asynchronous system architectures are sufficient. On the positive side, the asynchronous architectures based on approximate agreement yield systems that claim to be remarkably fault-tolerant to communication loss; for example, channels may remain operational with up to 20% communication packet loss. Interestingly, in the systems that we analyzed, Byzantine fault-tolerance is not specifically addressed other than for strategies to isolate asymmetric faults, as detailed in [1] . Hence, this aspect will be a part of our research agenda. Of particular interest is the behavior of the system during the time window required for asymmetric failure detection[2].

.

[1] As outlined in [7] the design of the voting strategies and control used in asynchronous systems are complex and non-trivial

[2] Up to 10 seconds of delay is required to confirm an asymmetric failure.

2 Case-study Architecture Review

The following sections present a summary of the architectures analyzed here-in. Further details of the detailed architectural mechanism are given [1].

2.1 Asynchronous Tripplex High Integrity Control

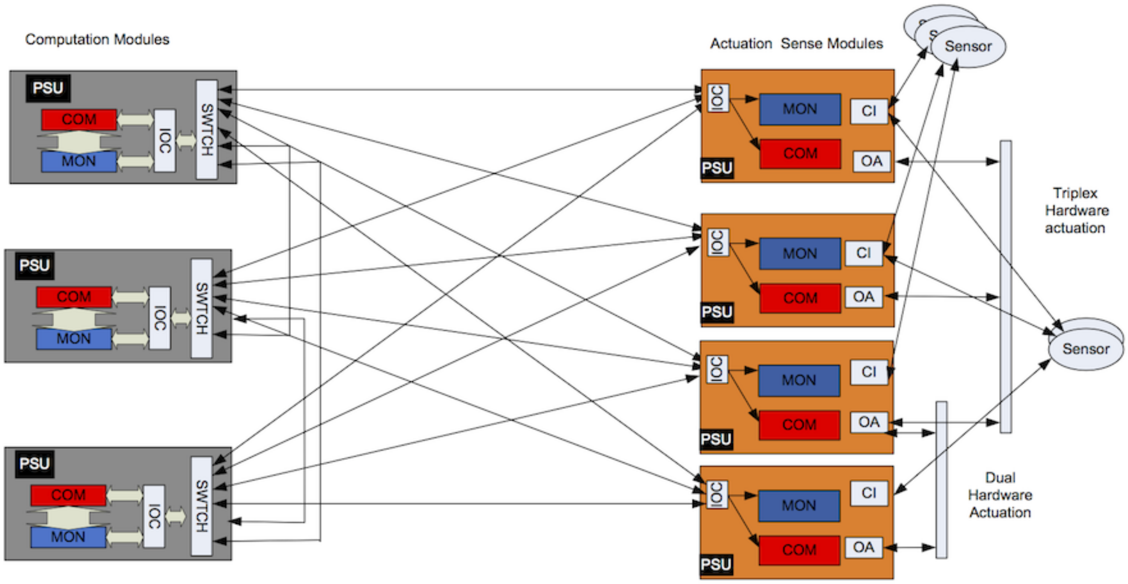


Figure 1 Asynchronous Three Channel Switched Ethernet Architecture

The first case-study architecture is illustrated in Figure 1. This system comprises three asynchronous computation modules (CM's) connected to four actuation and sense modules (ASMs). Each computation module communicates with the ASMs using a dedicated Ethernet network. The computation modules also communicate among themselves using the Ethernet networks.

All computation is done using self-checking hardware, incorporating a command and monitor computation lane within each CM. The monitor lane performs independent computation of the control algorithms and continuously monitors the output of the command lane. For each successful comparison, the monitor lane authenticates the validity of the commanded output by updating the values of the independent command signature and command confirmation heartbeat that are embedded within the each output message. The signature and heartbeat sequence[3] are validated by each ASM before the out message is used. A computational error by the command processor would result in an invalid signature or heartbeat value and the ASM would reject the message. To monitor the integrity of the Ethernet network transportation, the system also incorporates a wrap-back acknowledgement protocol. The ASMs also reflect an encoded function of each input message back to the sender for end-to-end integrity confirmation. The monitor lane of each control computer also monitors this reflected status of the previous command. If this reflected status is found to be erroneous, the monitor ceases authentication of the output command scheme and heartbeat. The lack of a valid signature and/or heartbeat signifies that the control channel is invalid.

[3] For any message to be considered valid, a heartbeat sequence counter embedded within the message is required to increment in a prescribed sequence.

The ASMs also use self-checking hardware with a command and monitor lane in each ASM that prevent failures in ASM processing from corrupting sensor input or causing hazardous hardware actuation.

In each ASM, a hybrid, mid-value selection function is used to select between the computation channel output commands. This selection is a function of how many of the computational output commands that an ASM receives are valid, with validity determined by the reception passing some in-line syntax tests that do not involve comparison among the command inputs. The function is implemented as follows:

- If all three command streams are valid, an ASM selects the mid-value of the three valid computation input streams.
- If only two of the computation input command streams are valid, the ASM uses the previous mid-value selection to supplement the two remaining streams, substituting the previous mid-value selection in-place of the missing or invalid command stream.
- If only one computation command input stream is valid the ASM uses this stream.

All tasking and communication within the system is implemented using a timed-asynchronous, model, i.e., each component independently executes a local periodic schedule of activity. The ASMs executes at the highest rate of the system, for example 80 Hz; whereas, the computations of the control computers are distributed across multiple rates, ranging from 80 to 1 Hz. Multiple ASMs operate cooperatively to drive the output actuation services, connected in dual and triplex configurations. The ASMs also process external and feedback sensor data and provide it to the CMs.

The system incorporates a number of strategies to ensure that the control computers remain aligned with respect to the commanded state. These strategies include the following:

- Internal integrator and discrete state equalization -- where each of the control computers continuously adjusts its state towards a fault-tolerant, mid-value function of the values from all operating lanes (which translates to majority voting for discrete signals)
- Communication asymmetry management -- where a control computer or ASM that is confirmed to be asymmetrically communicating, (i.e., a system component that is communicating with only a subset of the other system components) is isolated from influencing the group

For the initial investigation of actuation agreement, this document does not elaborate on these strategies; however, the details can be found in [1]. Our rationale is presented with the initial system modeling in Section 4.

2.2 Asynchronous BRAIN-based Ethernet Architecture

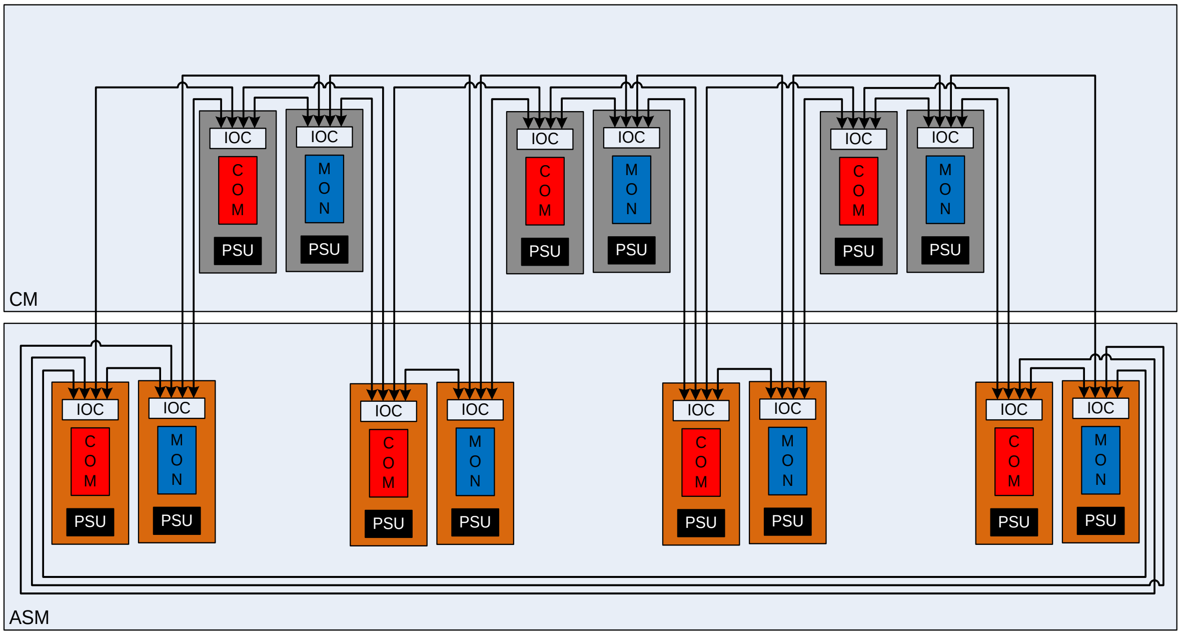


Figure 2 Ethernet Brain Based Case-study Architecture

The second case-study architecture is depicted in Figure 2. This system comprises the same components of the first system, but in place of the three switched Ethernet networks, a single Ethernet-based Braided Ring Availability Integrity Network (BRAIN) is used to connect the system components.

In this case-study architecture the asynchronous BRAIN 3.0 protocol is considered. This is a layered protocol that can be deployed on top of a standard Ethernet or a profiled Ethernet (e.g., ARINC 664) implementation. BRAIN 3.0 assumes that routing authentication and bandwidth fairness allocation is performed within the underlying Ethernet layer. For example, the underlying Ethernet layer can use fixed routing tables and configured bandwidth allocation. The BRAIN 3.0 protocol leverages these underlying properties to implement data integrity acceptance criteria that are a function of qualified, disjoint, data-distribution path mapping. That is, received messages are not accepted as valid unless multiple copies of the messages arrive from totally disjoint communication paths and the messages are bit-for-bit identical, with the disjoint communication paths being enforced by path mapping mechanisms in the underlying Ethernet layer. Using the enforced message routing strategy, we believe that the BRAIN 3.0 will yield a dual fault-tolerant (assuming non-colluding faults[4]) high-integrity message broadcast guarantee. A full summary of the BRAIN 3.0 protocol message routing and details of the data acceptances tests are given in [1].

Note that the role of the network in this second architecture is more integral to the system redundancy management arguments than the network of the initial case-study architecture. In the initial architecture, an end-to-end wrap-back protocol was implemented above the network to detect network data corruption. The BRAIN-based system also incorporates a number of high-level strategies to mitigate asymmetric communication failure.

In the BRAIN 3.0 architecture, the underlying communication system is intended to guarantee a Byzantine resilient data broadcast in the presence of up to and including two non-colluding faults. We believe that this property of guaranteed data broadcast consistency will greatly improve system

[4] Colluding faults are faults that act in support of each other.

performance while reducing the system complexity and overheads. This idea will be investigated as the two system architectures are modeled and compared.

Another area where the BRAIN 3.0 and the initial case-study architecture differ is the comparison of the command and monitor lane outputs. In the initial architecture, the control computer comparison is implemented in software, with the monitor implementing bounded comparison of the command lane output prior to authentication of the command transmission over the network. In the BRAIN 3.0-based architecture the command and monitor comparisons are performed with the network distribution function[5]. To produce congruent output, the COM and MON lanes of each self-checking pair rendezvous and synchronize using the dedicated link that connects them. Other than this synchronization between COM and MON, all other data flow of the BRAIN 3.0 architecture is asynchronous, with each pair executing a local periodic schedule of tasking and communication activity.

[5] For this scheme we assume that the output of the lanes is bit-for-bit identical. Should dissimilar processing hardware be employed the scheme assumes a fixed point arithmetic processing to ensure bit-for-bit lane congruency

2.3 Synchronous Two-tier Network Architecture

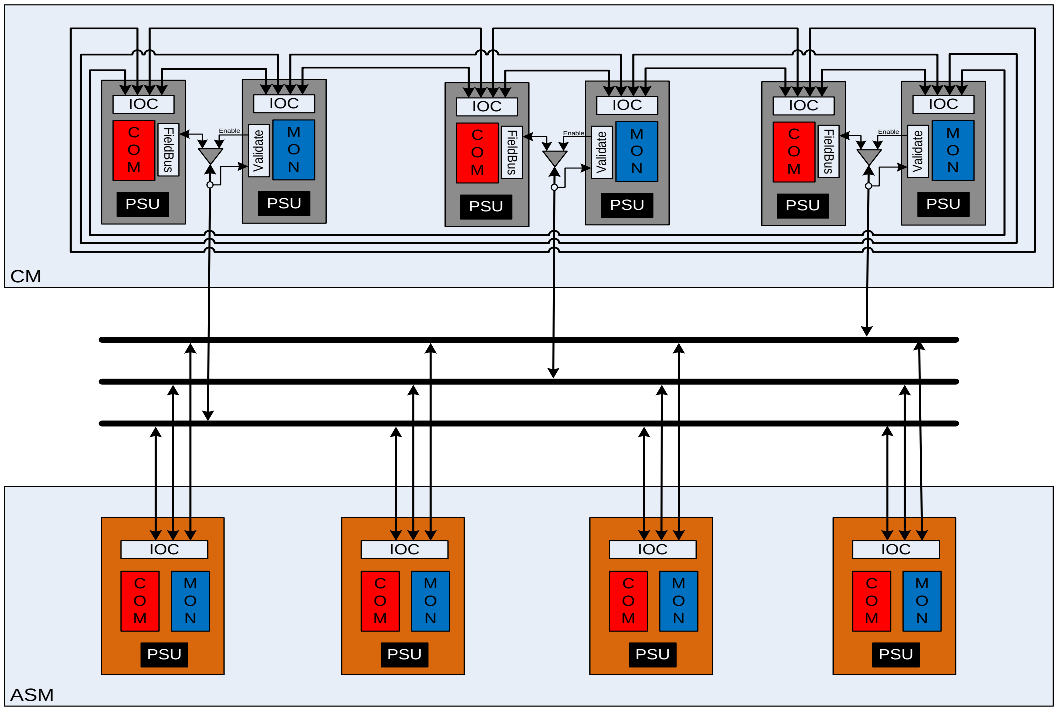


Figure 3 Two-tier Synchronous Network Case -study Architecture

The final case-study architecture is depicted in Figure 3. This system comprises the same components of the first system, but in place of the three switched Ethernet networks, a hybrid network architecture is deployed. In this architecture, a two-tier network architecture is considered. The control computers are fully interconnected and synchronized using a Time-Triggered Ethernet [7] network backbone. We assume that the quality of synchronization achieved in such a configuration will yield a synchronization precision of 25 µs.[6] To communicate with the ASMs, each control computer implements a dedicated access bus[7] connection using a typical access bus protocol, for example TTP [8]. To maintain system synchrony, the TTP access bus connections are also synchronized to the master Time-Triggered Ethernet schedule and timeline. Hence, this final system is globally synchronous with all system tasking and communication coordinated in accordance with the global Time-Triggered Ethernet timeline.

In this third architecture, a separate TTP network is dedicated to each control computer channel. Given this configuration, it is possible for asymmetric communication faults to manifest between the control computers and the ASMs. Therefore, we assumed that this third architecture deploys similar network management and asymmetric communication fault detection strategies as the first architecture. We further assumed that this synchronous architecture implements an end-to-end wrap-back protocol to mitigate network component integrity failures.

[6] A typical precision achieved in industrial configurations

[7] The Term "access bus" is used to denote the lower tier of a two-tier network.

3 A Discussion of the Case-study Architectures

3.1 Overview

The case-study architectures introduced in the previous section presents a number of different redundancy management policies. To facilitate a comparison of all three architectures, a single property is selected for formal examination. This selected property, colloquially termed *force-fight* denotes the level of command disagreement that exists across redundant actuation interfaces. Where multiple ASMs couple to a shared actuation interface, it is important that they maintain command congruency, since any discordance in command output may manifest as opposing forces applied to the actuation surface, which contribute unwanted surface stress that can in turn result in premature surface degradation and/or aging[8]. This is illustrated in Figure 4 below.

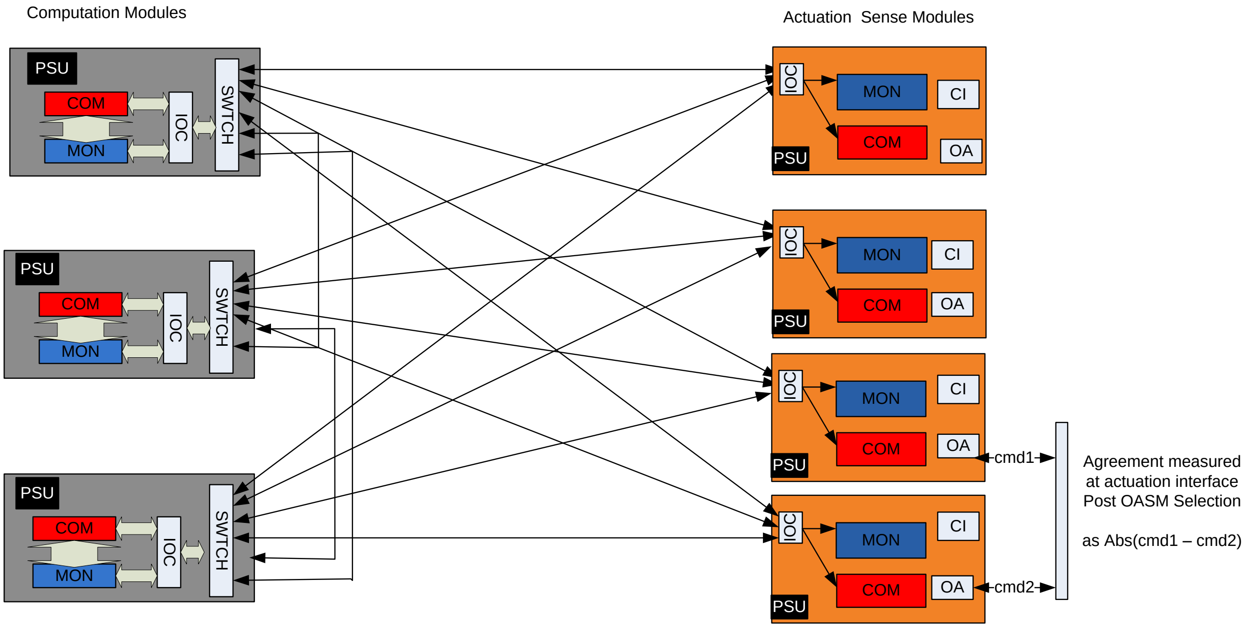


Figure 4 Actuation Force-Fight Instrumentation

Note that the level of agreement maintained at the actuation interface is determined solely by the distributed architecture redundancy management policies (including the degree of synchronization the redundancy management policy uses). In all of the three case-study architectures, the quality of actuation agreement is largely influenced by the emergent properties of the hybrid mid-value-selection function implemented within the ASMs, given possible asynchronous behavior of the commanded output streams from the control channels operating in normal and faulted conditions.

3.2 Assumed Failure Modes

The first stage in any architecture analysis is to define the assumed failure modes of the system components and communication. At first glance, the self-checking mechanisms of the computation and ASM hardware would normally lead to a fail-silent failure model. However, since the network hardware in the first and third architectures is not self-checking, this assumption would be invalid. These

[8] This is particularly important with composite airframe materials

architectures use a wrap-back based integrity check and isolation scheme that cannot contain all integrity violations; there exists an unlikely, but non-zero, probability of a network-induced corruption escaping the fault detection capability. Therefore, we assume that a single erroneous value may escape from the self-checking computer channel without detection. Persistent integrity errors are not assumed, since the encoded heartbeat protocol will cease command authentication on detection of the first error.

With respect to message distribution in the first and third architectures, we assume that some fault conditions may cause a compute channel to communicate asymmetrically with the ASMs. After review, we found that, in some systems, there is a significant fault detection lag in the systems' logic to mitigate such asymmetric communication. Therefore, under worst-case conditions, communication asymmetry may persist and contribute to output non-congruence before isolation takes place. We further assume that up to two compute channels may be faulty at the same time.

For the second (BRAIN-based) architecture, we assume the claimed fault model of the BRAIN 3.0 protocol. That is to say, faults are consistently observed by all the consuming components, and the network and computation functions are fail-silent with respect to integrity violation. Note that at the time of writing this report, this fault model had not been formally verified, although informal experiments based on model checking have demonstrated the assumed property.

4 Formally Modeling Actuation Agreement Using SAL

4.1 Formal Model Description

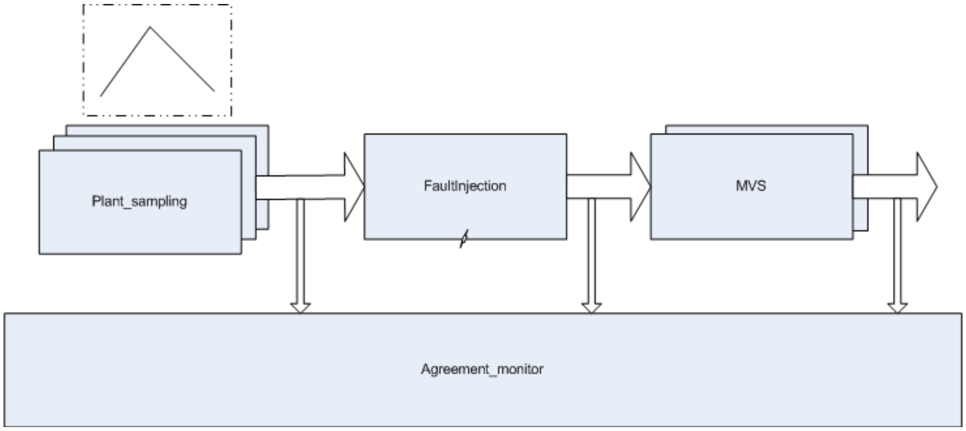


Figure 5 Formal Model High Level Structure

To make the initial analysis more tractable, we focused our initial modeling on the behavior of the output stage in isolation using the techniques proposed in [9]. From a high-level view, this model can be conceived as a system where the cross-channel state equalization functions result in perfect control-channel equalization. That is to say, we assume that there are no errors attributed to the internal state divergence of the control channels. Further, we simplified our initial analysis by analyzing the output behavior in an open-loop configuration to remove the complication of incorporating the continuous plant-model into the control feedback path[9]. A high-level pictorial representation of the simplified model is shown in Figure 5.

In this initial model, the controller can be viewed as a pass-though transfer function with a gain of 1. The dominant behavior within the model comprises the interaction of the asymmetric fault-injection with the asynchronous control execution and hybrid mid-value-selection logic of the ASMs.

The initial formal abstraction comprises the synchronous composition of the following components:

- The *plant* module produces a saw-tooth wave form and models the asynchronous sampling of this waveform by the three computer control channels.
- The *mvs* module models the hybrid mid-value-selection function of the ASMs.
- The *agreementmonitor* module comprises synchronous observer monitoring of key properties of interest.
- The *faultinjectionbus* module connects the *plant* and *mvs* modules. It distributes the computed output from the *plant* to the *mvs* and injects value and/or omission faults on selected signal paths.

[9] As part year 3 of our research we expect to integrate these details into the formal abstraction framework.

4.1.1 Asynchronous Interaction

In asynchronous systems modeling, time-dependent interaction is key and is necessary to understand the phase-dependent behavioral relationships. In the initial formal abstraction, this aspect of the system is explored using a bounded non-deterministic selection of the plant sampling point. It is captured in the synchronous_plant_sampling module of the mvs.sal file. The sampling phase of the first control channel is initialized to be non-deterministically assigned within the waveform period.

```
% tt1 is initially constrained to be greater dt and less than period
tt1 IN { z:REAL | z >= dt  AND z <= period};
```

The sampling phase of the two remaining channels is initialized to be set within a sample period of the value used for the first channel

```
% tt2 and tt3 are constrained to be within one interval of dt less than tt1
tt2 IN { z:REAL | z >= tt1 - dt  AND z <= tt1 AND z >= 0 AND z <= period };
tt3 IN { z:REAL | z >= tt1 - dt  AND z <= tt1 AND z >= 0 AND z <= period };
```

In every state transition, the sampling time of the first channel is increased by the sample period. However, since the plant waveform is only defined for a single-period boundary, the update of the channel sampling point is constrained to fold back into the plant waveform period once it reaches the interval within dt of the waveform period.

```
tt1' = IF tt1 <= period - dt THEN tt1 +  dt ELSE tt1 + dt - period ENDIF;
```

At each state transition, the sampling points of the second and third channels are bounded to a non-deterministically assigned interval within a sample period dt of the current tt1 value.

```
            tt2 in { z:real | z >= tt1  and z <= tt1 + dt};
            tt3 in { z:real | z >= tt1  and z <= tt1 + dt};
```

Note that this arrangement does not maintain a consistent sampling interval for the second and third channels. However, we believe that this is not necessary for the investigation of the ASM mid-value-selection behavior. This assignment also forces the exploration of all possible phase relationships of tt2 and tt3 with tt1. Hence, in the update of the sample time period, we do not need to introduce additional phase values to model phase drift.

To generate sample values, the plant simply returns the value of the test waveform for each of the sample time points. Note that at the bounds of the period waveform within channels 2 and 3 may lie beyond the scope of the sample waveform definition. To mitigate this, the sample times are simply reflected back into the waveform space by either subtracting the period for times greater than the period boundary or adding the period boundary for samples less than zero. This same structure is used for all sample channels

```
yp1 = waveform (tt1);
yp2 = IF tt2 <= period THEN waveform(tt2) ELSE waveform(tt2 - period) ENDIF;
yp3 = IF tt3 <= period THEN waveform(tt3) ELSE waveform(tt3 - period) ENDIF;
```

Note that verify the validity of the sampling assumptions the following test lemma was added

```
test1: LEMMA asynchronous_plant_sampling |- G(tt1 >= 0 AND tt1 <= period)
```

4.1.2 Modeling Synchronous Interaction

To model the time-triggered case study, we need to modify the channel timing alignment. This modification is done by constraining all three channels to a defined synchronization precision that is specified using the additional parameter *sync_precision*. This specification is captured in the *synchronous_plant_sampling* module.

```
% tt1 is initially constrained to be greater dt and less than period
  tt1 IN { z:REAL | z >= sync_precision  AND z <= period};

  % tt2 and tt3 are constrained to be within one interval of sync precision  interval  less than tt1
  tt2 IN { z:real | z >= tt1-sync_precision  AND z <= tt1};
  tt3 IN { z:real | z >= tt1-sync_precision  AND z <= tt1};
```

In the model we assume an achieved synchronized precision of 25 µs a typical value for industrial systems. For the model, the tighter agreement bound based on the synchronization precision was also added

```
sync_precision: real = 0.000025;
sync_e:  REAL = 1.0  * sync_precision * p2p
```

4.1.3 Fault-Injection

The *faultinjectionbus* modules are responsible for the distribution of the values from the plant modules to the two instances of the *mv*s module. These modules also introduce erroneous value and signal omission faults. Two fault injection scenarios are captured using the faultinjectionbus_IO23 and *faultinjectionbus_byzantine_channel* models. The structure of these modules is equivalent. The signals from the plant are input as *ypn* variables. These values are separately assigned to each of the output *mvs* channels that are *cn_xn* values. The *cn_bn flags* are used to validate the channel data. When set to FALSE, the data from the channel is considered invalid and the channel is omissive. In the *faultinjectionbus_IO23*, one of the *mvs* client modules is subjected to inconsistent omission failure of one or two computer control channels via the non-deterministic assignment of the validity flags for channels two and three.

```
faultinjectionbus_IO23: MODULE =
BEGIN
  INPUT
    yp1, yp2, yp3: REAL

  OUTPUT
    c1_x1, c1_x2, c1_x3: REAL,
    c2_x1, c2_x2, c2_x3: REAL,
    c1_b1, c1_b2, c1_b3: BOOLEAN,
    c2_b1, c2_b2, c2_b3: BOOLEAN

  DEFINITION
   % OUTPUT values are good
   c1_x1 = yp1; c1_x2 = yp2; c1_x3 = yp3;
   c2_x1 = yp1; c2_x2 = yp2; c2_x3 = yp3;

   % First Channel of MVS gets all good status values
   c1_b1 = TRUE; c1_b2 = TRUE;  c1_b3 = TRUE;

   % Second Channel of MVS  channels 2 and 3 are  inconsistently omissive
   c2_b1 = TRUE;
   c2_b2 IN { TRUE, FALSE};
   c2_b3 IN { TRUE, FALSE};

END;
```

In the *faultinjectionbus_byzantine_channel*, the Byzantine failure of one of the computer channels is modeled. These errors are coded to present inconsistent and/or erroneous values from one of the computer channels and from both of the mvs clients.

```
faultinjectionbus_byzantine_channel: MODULE =
BEGIN
  INPUT
```

```
    yp1, yp2,yp3: REAL

  OUTPUT
    c1_x1, c1_x2, c1_x3: REAL,
    c2_x1, c2_x2, c2_x3: REAL,
    c1_b1, c1_b2, c1_b3: BOOLEAN,
    c2_b1, c2_b2, c2_b3: BOOLEAN

  DEFINITION
   % Channels 1 and 3 have good values
   c1_x1 = yp1;  c1_x3 = yp3;
   c2_x1 = yp1; c2_x3 = yp3;

   % And Good Status Indication
   c1_b1 = TRUE;  c1_b3 = TRUE;
   c2_b1 = TRUE;  c2_b3 = TRUE;

   % Channel 2 is inconsistently omissive
   c1_b2 IN { TRUE, FALSE};
   c2_b2 IN { TRUE, FALSE};

   % And Byzantine
   c1_x2 IN { z:REAL | z >= 0 AND z <= p2p };
   c2_x2 IN { z:REAL | z >= 0 AND z <= p2p };

END;
```

For the BRIAN 3.0 architecture, a symmetric fault manifestation of up to two of the computer control channels is assumed[10]. This assumption is coded in the *faultinjectionbus_brain3* module. At each cycle of execution, the computer control outputs from Channels 2 and 3 to the first channel of the *mvs* are non-deterministically selected from TRUE and FALSE (good or faulty). The selected values of the Channel 2 and 3 faults are presented to the input of the second *mvs* channel.

```
    c2_b1' = TRUE;
    c1_b2' IN { TRUE, FALSE};
    c1_b3' IN {TRUE,FALSE};

    c2_b2' = c1_b2';
    c2_b3' = c1_b3';
```

4.1.4 MVS Evaluation

The mvs module calculates the mid-value selection output of the ASM. As described in [1], the selected output is a function of the number of input streams. When all three streams are valid, the mid value of the three inputs is used. When only two inputs are valid, the function selects using the two valid inputs and the previous mid-value selection as a substitute input for the missing stream, as illustrated in the code below.

```
% 3 INPUTs, 3 valid bits, OUTPUT 1 value
mvs :MODULE =
BEGIN
  INPUT
    x1, x2, x3 : REAL,
    b1, b2, b3 : BOOLEAN

  OUTPUT
    x : REAL

  INITIALIZATION
    x = 0

  TRANSITION
    % new mvs coasts when no good INPUT is available
    x' = midval(
        IF b1' THEN x1' ELSIF b2' AND not(b3') THEN x2' ELSIF b3' AND not(b2') THEN x3' ELSE x ENDIF,
        IF b2' THEN x2' ELSIF b1' AND not(b3') THEN x1' ELSIF b3' AND not(b1') THEN x3' ELSE x ENDIF,
        IF b3' THEN x3' ELSIF b1' AND not(b2') THEN x1' ELSIF b2' AND not(b1') THEN x2' ELSE x ENDIF);
END;
```

For the selected inputs, the module calls a function returning the mid-value selection.

```
     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;
```

[10] As discussed earlier, it is emphasized that at the time this property has not been formally verified.

4.1.5 The Agreement Monitor

The *agreementmonitor* implements a synchronous observer [9] that monitors key points of interest within the model.

The obvious instrumentation is the value difference observer between the two *mvs* channels. The expected bound of the agreement is dependent on the case-study architectural policies. Hence, the agreement monitor module includes dedicated flags for each of the agreement thresholds.

```
flag_async_bounded_agreement = (mvs_1_x - mvs_2_x <=  error AND mvs_2_x - mvs_1_x <= error);
flag_sync_bounded_agreement = (mvs_1_x - mvs_2_x <=  error AND mvs_2_x - mvs_1_x <= error);
```

4.1.6 Model Composition

The modules described in the previous section were composed synchronously to support the evaluation of the cases-study architectures.

- *systemmonitor_IO23_asynchronous*: Represents the asynchronous system with inconsistent omission failure of up to two compute channels.
- *systemmonitor_byzantine_channel_asynchronous*: The prevous architecture with a Byzantine failure of a single compute channel.
- *systemmonitor_IO23_synchronous*: Represents the time-triggered synchronous system with inconsistent omission failure of up to two compute channels.
- *systemmonitor_brain3_asynchronous*: Represents the asynchronous system with the symmetric fault model of the BRAIN 3 architecture.

To further validate the abstraction, we composed additional system configurations that included a transient failure of the sampled waveform. This inclusion violated the expected rate of change assumption of the sawtooth waveform. Therefore, we wanted to check if the use of such a waveform would result in counter examples. These scenarios are captured in the *systemmonitor_IO23_asynchronous_wt*. The modified sample waveform code is shown below.

```
waveform_wt( t : REAL,tr : REAL): REAL =
           IF t <= period/2 THEN  t
           elsif t >= tr  then 0
           ELSE
             1.0 - ((t - 1.0) )
           ENDif;
```

The position of the transient was non-deterministically assigned to lie within the plant period.

```
           tr IN {z:REAL | z >=0  and z <= period};
     % tt1 is initially constrained to be greater dt and less than period
```

4.1.7 Proving Agreement Properties

We used the formal models of the study architectures to explore the fault-tolerance and agreement properties of the case-study system. These initial experiments were performed using the *sal-inf-bmc* model checker. Initially, the depth of the exploration was set to be larger than the waveform period. The calculation of the required depth is dependent on the model structure. In this example, the periods of the mvs, fcm and the sampled plant waveform are harmonically related, therefore exploring to the depth of the plant waveform is sufficient.

Following these informal experiments, we used the k-induction capability of the *sal-inf-bmc* model checker to prove agreement. Using this initial abstraction both theorems were proven to be true.

Using these additional lemmas, we could prove the bounded agreement using a depth k= 2. For the asynchronous architecture with two omission faults, the level of agreement corresponded to the temporal skew of the computer channel sampling. That is to say, the level of agreement was determined by the maximum rate of change of the plant waveform and the corresponding maximum divergence that may occur over the sampling period.

```
expected_error :      REAL = 1.0  * dt * p2p
```

For the asynchronous architecture, the Byzantine fault-scenario also was explored. Interestingly, under the Byzantine failure scenario, the level of bounded agreement was equivalent to the omission scenarios.

For the synchronous architecture, the level of expected error was reduced to correspond to the synchronous system precision. Using k-induction, this level of expected behavior was found to be a correct bound.

```
sync_e:  REAL = 1.0  * sync_precision * p2p ;
```

We repeated this process with the Byzantine channel present and once again did not observe a violation of the agreement bound property proving.

Finally, for the BRAIN-based asynchronous architecture, we instrumented the agreement monitor to monitor exact agreements; (i.e., zero error). This property also was proven using an induction depth of k = 1.

4.1.8 Additional Model Validation Experiments

To further validate the asynchronous model, we lowered the threshold of bounded agreement to 0.99999 of the expected value. This threshold returned a counter example as expected. In addition, the models with the transient disturbance injected into the plant waveform also returned counter examples as the rate of input change under the transient scenarios increased beyond the expected maximum assumed skew.

4.2 Discussion of Initial Model and Findings

The initial formal investigation of the Phase 2 case-studies yielded a number of lessons and, in some cases, lessons re-learned. In the asynchronous architecture, the bound of agreement at the actuation interfaces is solely determined by the asynchronous plant sampling. With our simplified saw-tooth plant waveform this corresponded to the delta change in the plant waveform that occurs during the period duration of the sampling task. Using the *sal-inf-bmc* model checker we were able to prove this bound of agreement, for fault scenarios comprising one or two inconsistent omissive faults. Interestingly the presence of a single Byzantine fault did not degrade this level of agreement. Therefore it may be argued that this class of architecture is not vulnerable to Byzantine failure[11]. Although, this finding was initially surprising it is in line with our definitions[10] of Byzantine faults and Byzantine failure:

- Byzantine fault: a fault presenting different symptoms to different observers.
- Byzantine failure: the loss of a system service due to a Byzantine fault in systems that require consensus.

Using the above, if the level of disagreement due to the asynchronous sampling is sufficient for performance, then additional strategies for exact agreement are not required. This finding helps qualify our findings from the asynchronous case-study of the first year [2]. From this study, we concluded that the overhead required for exact agreement may be too expensive for practical use. This new finding further helps to illuminate the differences between the asynchronous and synchronous design mindsets. If the system can perform with such levels of inexact agreement, it easier to understand why system architects with an asynchronous system design preference resist any suggestion to increase the level of channel coupling to achieve a tighter bound of agreement if it is not be needed. Such strategies allow them to avoid common-mode influence from such agreement services that can increase system brittleness. That being said, as illustrated [11], developing voting and fault-isolation strategies with inexact agreement can be relatively complex, requiring extensive knowledge of the system dynamics. The increase in channel coupling from cross-lane equalization also needs also to be considered and analyzed under normal and failure modes.[12] Consequently, validating the effectiveness of such strategies is also non-trivial.[13] In addition, arguing platform fault-tolerant properties independent of the hosted application is very difficult within an asynchronous architecture, since the plant dynamics and asynchronous tasking rates are closely coupled to the fault-detection thresholds and performance.

For the time-triggered synchronous architecture, the platform fault-tolerant properties are simpler to establish, since they are dominated by the system precision and less influenced by tasking rate and plant dynamics. In our study, the mid-value-selection also constrained the influence of a single Byzantine fault to be within the agreement bound supported by the synchronous precision. Obviously, the synchronization services underpinning such a system must also be validated for Byzantine fault-tolerance.

Finally, it is interesting to note that the consistent broadcast guarantee[14] of the BRAIN may achieve exact agreement in either synchronous or asynchronous operational modes. This is an interesting option for such architectures. However, the impact of transient errors also needs to be assessed. The extended hierarchal agreement services detailed in [12] may be one option to increase the transient robustness.

[11] Alternatively it may be argued that the asynchronous sampling may itself be considered equivalent to a Byzantine fault.

[12] Given the complexity and application-specific nature of equalization schemes, such analysis may be more complex than that of a fault-tolerant synchronization service.

[13] Fortunately, in the case-study architecture the majority of the fault-detection is implemented at the source via the comparison of the Command and Monitor lanes. Hence in such systems, the fault detection threshold calculation may be simpler to establish.

[14] Yet to be formally verified

4.3 Initial Model Checking Performance and findings

The model checking performance also was acceptable for all models, with proofs completing within a few seconds. We believe that the non-deterministic assignment of the initial plant sampling phase-offsets facilitates the exploration of phase-related emergent behavior. The performance of the bounded model checker was satisfactory, yielding results within a few seconds. In addition, in this initial model, deriving a proof from the model was straight forward, since it did not require the generation of any additional auxiliary lemmas.

However, in this initial model the synchronous composition of the *mvs* modules is a deficiency and the synchronous abstraction may miss effects resulting from the asynchronous boundary between the *mvs* modules and their respective tasking rates. In addition, although this initial model is sufficient for the open loop exploration, we are uncertain how to evolve this model to integrate the closed-loop control and plant models model. To ameliorate these shortcomings, we explore an alternative abstraction in the next section.

4.4 An Alternative Timeout Automata Based Abstraction

To address the issues discussed in the previous section, we developed an alternative abstraction using the timeout automata[13]. We hope that this abstraction will enable the impact of the asynchronous interaction of the *mvs* systems to be analyzed. We further hope that this revised abstraction will facilitate for the integration of closed- loop control and plant behavior into the model.

The initial timeout-automata based abstraction contains very similar components to those described in the previous section. Hence the details of the subcomponents are not elaborated in detail below. Instead we concentrate on the differences related to the capture of the asynchronous tasking using the timeout automata based framework.

4.4.1 Clock Module

A central component of the revised model is the clock module. This central module is responsible for incrementing the global time. The progression of time is governed by this module. Each system component provides an input to this module (via the *_timeout* signals). This input corresponds to the time of the respective components next timed action (i.e., the value of its local timeout). The clock module evaluates the global array of timeout events and advances a global *time* variable to the lowest value in the global timeout array.

```
        % Clock module: advance time to min(fcm_timeout1, fcm_timeout2, fcm_timeout3, mvs_timeout1, mvs_timeout2)
        %
        clock: MODULE =
          BEGIN
           INPUT
             fcm_timeout1, fcm_timeout2, fcm_timeout3: TIME,
             mvs_timeout1, mvs_timeout2: TIME
           OUTPUT
             time: TIME
           INITIALIZATION
             time = 0;
           TRANSITION
            [ time < fcm_timeout1 AND time < fcm_timeout2 AND time < fcm_timeout3 AND time < mvs_timeout1 AND time < mvs_timeout2
-->
                    time' IN { t: TIME | t <= fcm_timeout1 AND t <= fcm_timeout2 AND t <= fcm_timeout3
                                         AND t <= mvs_timeout1 AND t <= mvs_timeout2
                                         AND (t = fcm_timeout1 OR t = fcm_timeout2 OR t = fcm_timeout3 OR t = mvs_timeout1 OR t =
mvs_timeout2) };
            ]
           END;
```

4.4.2 Source Module

This second abstraction also introduces a source module to represent the stimulus of the system. This enables improved modeling of the *fcm* sampling[15]. This module derives a period count from the time and this is value is subtracted from the time value before calling the waveform function (because the waveform function is only defined for a single plant period).

```
       %
       % Source module: at time t, the output is x = waveform(t - k * period)
       % where period * k <= t < period * (k + 1)
       %
       source: MODULE =
         BEGIN
           INPUT time: TIME
           LOCAL period_counter: INTEGER
           OUTPUT x: REAL
         DEFINITION
           period_counter IN { n: INTEGER | n * plant_period <= time AND time < (n+1) * plant_period };
           x = waveform(time - period_counter * plant_period);


         END;


       waveform(t: REAL): REAL =
```

[15] Note that this is a simplification from real system, where the input sampling is performed in the ASM modules.

```
IF t < 0 OR t > plant_period THEN 0
ELSIF t <= plant_period/2 THEN A * t
ELSE amplitude - A * (t - plant_period/2)
ENDIF;
```

4.4.3 FCM Module

The *fcm* is very similar to the *fcm* of the initial model. In comprises a very simple state transition. When the value of *time* equals the *timeout* value the fcm updates the sample value *y*, to the current value of the source *x*. In this initial model *y* is initialized to 0 and *timeout* is initialized to the 1st value. Due to the synchronous system composition (see section 4.1.6) the *fcm* module also defines an empty else transition.

```
faulty_fcm: MODULE =
   BEGIN
     INPUT
       time: TIME
     OUTPUT
       timeout: TIME,
       y: REAL
     INITIALIZATION
       timeout IN { t: TIME | 0 <= t AND t < fcm_period };
     TRANSITION
       [ time = timeout -->
            timeout' IN { t: TIME | time + epsilon <= t };
            y' IN { x: REAL | true };
      []
         ELSE -->
       ]
   END;
```

Note: The initial value pre_y has been included to support the proof by induction. This is discussed later with the proof.

4.4.4 MVS Module

The *mvs* module is very similar to the previous model. When the time is equal to the timeout function the modules updates the *mvs* output using the *midval* function. Similar to the initial abstraction, the *mvs* calculation is a function of the number of valid inputs, with input validity been denoted by Boolean flags *bn* for input each channel. The model also defines an empty *ELSE* transition. This is to support the synchronous composition of this module with other system modules (see section 4.1.6).

```
mvs_period: TIME = 0.05;

  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
        y1, y2, y3: REAL,
        b1, b2, b3: BOOLEAN,
        time: TIME
      OUTPUT
        timeout: TIME,
        mvs: REAL

      INITIALIZATION
        timeout IN { t: TIME | 0 <= t AND t < mvs_period };
        mvs = 0;

      TRANSITION
        [ time = timeout -->
             timeout' = time + mvs_period;
             mvs' = midval( IF b1' THEN y1' ELSIF b2' AND NOT(b3') THEN y2' ELSIF b3' AND NOT(b2') THEN y3' ELSE mvs ENDIF,
                            IF b2' THEN y2' ELSIF b1' AND NOT(b3') THEN y1' ELSIF b3' AND NOT(b1') THEN y3' ELSE mvs ENDIF,
                            IF b3' THEN y3' ELSIF b1' AND NOT(b2') THEN y1' ELSIF b2' AND NOT(b1') THEN y2' ELSE mvs ENDIF);
       []
          ELSE -->
        ]

    END;
```

4.4.5 Fault Injection

This module also incorporates fault injection. In the initial experiments this However, due to the difficulties encountered with the agreement proof (see later discussion), the faults are set to be inactive.

```
% Fault model: sampler 2 is faulty so c1_b2 and c2_b2 can true or false
%
fault_injection : MODULE =
  BEGIN

  OUTPUT
  c1_b1, c1_b2, c1_b3: BOOLEAN,
  c2_b1, c2_b2, c2_b3: BOOLEAN

  DEFINITION

  c1_b1 = true;
  c1_b2 IN { true, false } ;
  c1_b3 = true;

  c2_b1 = true;
  c2_b2 IN { true, false };
  c2_b3 = true;

  END;
  c2_b3 = true;

  END;
```

4.4.6 System Composition

The system composition is depicted below. The *mvs* and *fcm* modules are first synchronously composed into a *system.* This system is then asynchronously composed with the synchronous composition of the *clock* and *source* modules. It is crucial to compose the clock module asynchronously with the system. Otherwise, there would be a clear deadlock.

```
%
   % Full system:
   % - synchronous composition of these components
   %
   system: MODULE =
     (RENAME timeout TO fcm_timeout1, y TO y1, pre_y TO pre_y1 IN fcm)
     || (RENAME timeout TO fcm_timeout2, y TO y2 IN faulty_fcm)
     || (RENAME timeout TO fcm_timeout3, y TO y3, pre_y TO pre_y3 IN fcm)
     || fault_injection
     || (RENAME b1 TO c1_b1, b2 TO c1_b2, b3 TO c1_b3, timeout TO mvs_timeout1, mvs TO mvs1 IN mvs)
     || (RENAME b1 TO c2_b1, b2 TO c2_b2, b3 TO c2_b3, timeout TO mvs_timeout2, mvs TO mvs2 IN mvs);
```

With this composition two types of transitions are taken alternatively:

- 'transition in the physical model (clock + source)'
- 'transitions in the system (samplers + voters)'.

In the first type of transition, time advances and x is updated. In the second type of transition, time and x are fixed but at least one of the samplers*/mvs/fcm* modules makes a discrete step. On every transition of clock, 'time' is increased where as on every transition of system 'time' stays unchanged. Therefore the clock and system can't be composed synchronously. However, the clock and source can be composed synchronously given that the output variable of the source module changes when 'time' changes.

4.4.7 Investigating and Proving the System Agreement Properties

This second model presented many more difficulties than the course abstraction of the first model. Firstly, given that the initial phase of the plant waveform is set to 0 in this second abstraction, the bounded model checking experiments require a significant depth of exploration. The required depth is calculated by the plant waveform period (*2s*) divided by the highest sampling rate (*0.05s*), i.e., a value of 40. However, since the mixed asynchronous/synchronous composition of the system components requires two steps to increase the value of the clock value, this value needs to be doubled to 80. To cross-check this value, a simple time-based lemma was added to the model to validate time against the depth of model exploration.

```
t: LEMMA full |-G(time < plant_period);
```

However, given this required depth of examination, the bounded model checking experiments were relatively slow, requiring approximately one hour of compute time on a high performance processor. Secondly, a significant effort was required to produce the auxiliary lemma to support a k-induction based proof of the agreement (even under no-fault scenarios).

The first set of invariants constrains the value of time and the timeout parameters. The value *time* is constrained to be less than the current *timeout* and the *timeout* is, in turn, bounded to be less than the current *time* plus the respective period. These invariants are provable by an induction depth of 1.

```
%
fcm_timeout_bounds1: LEMMA full |- G(time <= fcm_timeout1 AND fcm_timeout1 <= time + fcm_period);

fcm_timeout_bounds2: LEMMA full |- G(time <= fcm_timeout2 AND fcm_timeout2 <= time + fcm_period);

fcm_timeout_bounds3: LEMMA full |- G(time <= fcm_timeout3 AND fcm_timeout3 <= time + fcm_period);

mvs_timeout_bounds1: LEMMA full |- G(time <= mvs_timeout1 AND mvs_timeout1 <= time + mvs_period);

mvs_timeout_bounds2: LEMMA full |- G(time <= mvs_timeout2 AND mvs_timeout2 <= time + mvs_period);
```

The second set of invariants constrains the fcm sample values (*y)*. These are constrained to be within a fixed delta of stimulus source waveform output (*x*), where the size of the delta is derived from the fcm sampling period and the stimulus amplitude.

```
%
% Provable by induction at depth 1
%
sampling_error1: LEMMA
  full |- G(x - y1 <= A * (time - (fcm_timeout1 - fcm_period)) AND y1 - x <= A * (time - (fcm_timeout1 - fcm_period)));

sampling_error2: LEMMA
  full |- G(x - y2 <= A * (time - (fcm_timeout2 - fcm_period)) AND y2 - x <= A * (time - (fcm_timeout2 - fcm_period)));

sampling_error3: LEMMA
  full |- G(x - y3 <= A * (time - (fcm_timeout3 - fcm_period)) AND y3 - x <= A * (time - (fcm_timeout3 - fcm_period)));
```

The next set of lemmas constrain the behavior of two successive samples; that is, a previous sample is constrained to be within a fixed delta of the current sample, where the delta once again is a function of the waveform amplitude and the sampling

```
%
% Bound on the difference between two successive samples
% - provable by induction at depth 1, using sampling_error<i> as a lemma
%
pre_sampling_delta1: LEMMA
  full |- G(y1 - pre_y1 <= A * fcm_period AND pre_y1 - y1 <= A * fcm_period);

pre_sampling_delta2: LEMMA
  full |- G(y2 - pre_y2 <= A * fcm_period AND pre_y2 - y2 <= A * fcm_period);

pre_sampling_delta3: LEMMA
  full |- G(y3 - pre_y3 <= A * fcm_period AND pre_y3 - y3 <= A * fcm_period);
```

The set of auxiliary lemmas below constrains the sampling differences across the *fcm* channels. These are pair-wise that constrain the channel samples (y) to be within the fixed delta derived from the waveform amplitude and sampling rate. The invariants can be proven using the respective period channel's sampling error.

```
%
% Bound on the difference betweeen two sampling channels
% - to prove sampling_delta(i,j) use induction at depth 1, with sampling_error<i> and sampling_error<j> as lemmas
%
sampling_delta12: LEMMA
  full |- G(fcm_timeout1 >= fcm_timeout2 => y1 - y2 <= A * (fcm_timeout1 - fcm_timeout2)
                                        AND y2 - y1 <= A * (fcm_timeout1 - fcm_timeout2));

sampling_delta21: LEMMA
  full |- G(fcm_timeout2 >= fcm_timeout1 => y1 - y2 <= A * (fcm_timeout2 - fcm_timeout1)
                                        AND y2 - y1 <= A * (fcm_timeout2 - fcm_timeout1));

sampling_delta13: LEMMA
  full |- G(fcm_timeout1 >= fcm_timeout3 => y1 - y3 <= A * (fcm_timeout1 - fcm_timeout3)
                                        AND y3 - y1 <= A * (fcm_timeout1 - fcm_timeout3));

sampling_delta31: LEMMA
  full |- G(fcm_timeout3 >= fcm_timeout1 => y1 - y3 <= A * (fcm_timeout3 - fcm_timeout1)
                                        AND y3 - y1 <= A * (fcm_timeout3 - fcm_timeout1));

sampling_delta23: LEMMA
  full |- G(fcm_timeout2 >= fcm_timeout3 => y2 - y3 <= A * (fcm_timeout2 - fcm_timeout3)
                                        AND y3 - y2 <= A * (fcm_timeout2 - fcm_timeout3));

sampling_delta32: LEMMA
  full |- G(fcm_timeout3 >= fcm_timeout2 => y2 - y3 <= A * (fcm_timeout3 - fcm_timeout2)
                                        AND y3 - y2 <= A * (fcm_timeout3 - fcm_timeout2));
```

The final two lemmas constrain the behavior for the *mvs* selection under no fault scenarios. Each is defined to be the mid-value functions of the current active sample or the previous sampled value. Each is provable at induction depth 1 using the *fcm_timeout* auxiliary lemmas introduced above.

```
mvs_invar1: LEMMA full |- G(mvs1
               = midval(IF fcm_timeout1 - fcm_period <= mvs_timeout1 - mvs_period THEN y1 ELSE pre_y1 ENDIF,
                        IF fcm_timeout2 - fcm_period <= mvs_timeout1 - mvs_period THEN y2 ELSE pre_y2 ENDIF,
                        IF fcm_timeout3 - fcm_period <= mvs_timeout1 - mvs_period THEN y3 ELSE pre_y3 ENDIF));
```

```
   mvs_invar2: LEMMA full |- G(mvs2
= midval(IF fcm_timeout1 - fcm_period <= mvs_timeout2 - mvs_period THEN y1 ELSE pre_y1 ENDIF,
                                 IF fcm_timeout2 - fcm_period <= mvs_timeout2 - mvs_period THEN y2 ELSE pre_y2 ENDIF,
                                 IF fcm_timeout3 - fcm_period <= mvs_timeout2 - mvs_period THEN y3 ELSE pre_y3 ENDIF));
```

Using the *mvs_invar1*, *mvs_invar2* ,*sampling_delta12* , *sampling_delta13* , *sampling_delta23*, *sampling_delta21* , *sampling_delta31* , *sampling_delta32* , *pre_sampling_delta1*, *pre_sampling_delta2*, *pre_sampling_delta3* auxiliary lemmas, the agreement property can be proved at an induction depth of 1.

```
agreement: THEOREM full |- G(mvs1 - mvs2 <= error AND mvs2 - mvs1 <= error);
```

4.5 Investigating Faults with the Timeout Automata Based Abstraction

The previous model facilitates the proof of agreement in the fault-free case. However, the proof does not hold under fault conditions. In addition, the depth of required exploration in the previous model precludes practical interactive model checking, since each run requires over 2 hours of execution time. Therefore, the model was modified to incorporate a non-deterministically selected initial plant waveform phase offset. To do this, an additional global *start_phase_type* and an associated unbounded s*tart-phase* constant was added to the context

```
start_phase_type : TYPE =  { t: TIME | 0 <= t  AND t < plant_period - fcm_period };
start_phase : start_phase_type;
```

The initialization value of time of the c*lock*, *mvs*, and *fcm* modules were then defined as a function of the *start_phase* constant. Similarly, the initial sample (*y*), pre-sample (*pre_y)*, and *mvs* values were initialized to be the value of the source waveform at the time of the *start_phase* value.

```
mvs module …
timeout IN { t: TIME | start_phase <= t AND t < start_phase + mvs_period };
        mvs = waveform(start_phase);


timeout IN { t: TIME |  start_phase <= t  AND t < start_phase + fcm_period };
        y = waveform(start_phase);
        pre_y = waveform(start_phase);
```

With this modification, the depth of model exploration and search time can be greatly reduced as the depth of exploration can be reduced to 4. This value accommodates for the two steps incurred from the asynchronous composition of (*clock* || *source*) and two discrete steps for '*system*'[16].

4.5.1 Proving Agreement with Faults

To support the proof of agreement using the timeout automata model additional auxiliary lemmas were required. The following lemmas constrain the values of the samples on channels 1 and 3 to be a function of the source waveform at the time of reference

```
y_invar1: LEMMA
  full |- G(fcm_timeout1 >= fcm_period => y1 = waveform(fcm_timeout1 - fcm_period - n1 * plant_period));


y_invar3: LEMMA
  full |- G(fcm_timeout3 >= fcm_period => y3 = waveform(fcm_timeout3 - fcm_period - n3 * plant_period));
```

Similarly the previous samples of channels 1 and 3 are also constrained

```
% depth 1, lemma: y_invar1
pre_y_invar1: LEMMA
  full |- G(fcm_timeout1 >= 2 * fcm_period => pre_y1 = waveform(fcm_timeout1 - 2 * fcm_period - pre_n1 * plant_period));

% depth 1, lemma: y_invar3
pre_y_invar3: LEMMA
  full |- G(fcm_timeout3 >= 2 * fcm_period => pre_y3 = waveform(fcm_timeout3 - 2 * fcm_period - pre_n3 * plant_period));
```

The initial values of the sample and pre-sample values are also constrained

```
y_init1: LEMMA
  full |- G(fcm_timeout1 < fcm_period => y1 = 0);

y_init3: LEMMA
  full |- G(fcm_timeout3 < fcm_period => y3 = 0);


pre_y_init1: LEMMA
  full |- G(fcm_timeout1 < 2 * fcm_period => pre_y1 = 0);

pre_y_init3: LEMMA
   full |- G(fcm_timeout3 < 2 * fcm_period => pre_y3 = 0);
```

Each of the above invariants is provable using an induction depth of 1.

Additionally the lemmas constraining the bounds on the *mvs* values were revised as shown below

```
mvs_bounds1: LEMMA full |- G((z11 <= mvs1 AND mvs1 <= z13) OR (z13 <= mvs1 AND mvs1 <= z11));
mvs_bounds2: LEMMA full |- G((z21 <= mvs2 AND mvs2 <= z23) OR (z23 <= mvs2 AND mvs2 <= z21));
```

These can be proved using an induction depth of 2 in conjunction with the *fcm_timeout* auxiliary lemmas presented in the previous section.

Agreement can then be proved at an induction depth of 1 using all of the above lemmas in conjunction with the *fcm_timeout* lemmas of the previous section.

[16] However it is emphasized that this assumes depth assumes the harmonic relationship among the *plant*, *mvs* and *fcm* period. If this property is not valid, this depth may need to be revised and increased.

4.6 Run Scripts and Model Source Files

Source files and run scripts for the models described in the previous sections are posted at the NASA DASHlink site AFCS – Distributed Systems (https://c3.nasa.gov/dashlink/projects/79/).

Section	Model	Run File
4.1.1 - 4.1.7	mvs.sal	run_mvs.sh
4.4.1 - 4.4.74	mvs_with_timout1.sal	run_mvs_with_timeout1.sh
4.5.1	mvs_with_timeouts3.sal	run_mvs_with_timeouts3.sal

5 Mathematical Analysis of Mid-Value-Selection

This section presents a mathematical analysis of the actuation force-fight, with the intention of formally bounding the level of actuation agreement. Figure 6 illustrates three 1Hz sinusoidal CM outputs where the third channel drops off when time equals 3.0 seconds. In this example $c_1 = 1, c_2 = 0.9, c_3 = 0.75, \varphi_1 = 0, \varphi_2 = 0.1, \varphi_3 = 0.2$.

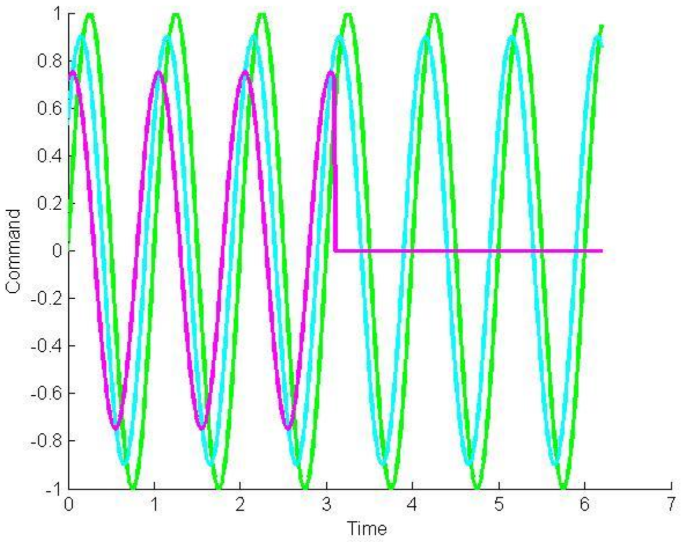


Figure 6 CM Sinusoidal Outputs

Let, $c_i \sin(x + \varphi_i)$ be the output from the ith channel, where $0 < c_i \leq 1, 0 \leq \varphi_i \leq 1$. An intersection between two channels will occur when $c_i \sin(x + \varphi_i) = c_j \sin(x + \varphi_j)$. Using standard trigonometric identities, both sides of the equation can be expanded to

$$c_i \sin(\varphi_i) \cos(x) + c_i \cos(\varphi_i) \sin(x) = c_j \sin(\varphi_j) \cos(x) + c_j \cos(\varphi_j) \sin(x) \tag{1}$$

Two special cases emerge. One, where $c_i = c_j, \varphi_i = \varphi_j$, the commands are identical and intersect everywhere. The other, where $c_1 \neq c_2, \varphi_1 = \varphi_2$, the commands intersect at $n\pi, n = 0, 1, 2, \ldots$ with a zero value. For all other cases, we can simplify (1),

$$\sin(x)(c_i \cos(\varphi_i) - c_j \cos(\varphi_j)) = \cos(x)(c_j \sin(\varphi_j) - c_i \sin(\varphi_i)),$$

and find the points of intersection at

$$\{x_{ij}\} = \left\{tan^{-1}\left(\frac{c_j \sin(\varphi_j) - c_i\, sin(\varphi_i)}{c_i\, cos(\varphi_i) - c_j\, cos(\varphi_j)}\right) + n\pi, n = 0,1,2,\ldots\right\} \tag{2}$$

Note, these values are in radians and need to be converted back to time by dividing the results by 2π. From (2) we have three sets of intersections, in seconds:

$$\{x_{12}\} = \{0.1744 + n/2, n = 0, 1, 2, \ldots\}$$
$$\{x_{13}\} = \{0.1191 + n/2, n = 0, 1, 2, \ldots\}$$
$$\{x_{23}\} = \{0.0566 + n/2, n = 0, 1, 2, \ldots\}$$

The model for the mid-value select is to use the median of the three channels if all are available. If only two channels are available, replace the missing channel with the previous mid-value and recalculate the median. If only one channel is available, the use that channel without any modification.

The mid-value select algorithm, described above, is implemented in MATLAB notation as:

```
function mv=midval(y1,y2,y3)
  if (y1 <= y2)
   if (y2 <= y3)
     mv=y2;
   elseif (y1 <= y3)
     mv=y3;
   else
     mv=y1;
   end
 else
   if (y1 <= y3)
     mv=y1;
   elseif (y2 <= y3)
     mv=y3;
   else
     mv=y2;
   end
 end
end
```

It is possible to bound this disagreement in closed form. The main disagreement is due to the difference between sinusoids. Recall, $c_i \sin(x + \varphi_i)$ is the output from the ith channel, where $0 < c_i \leq 1, 0 \leq \varphi_i \leq 1$. The difference between these channels is

$$c_i \sin(x + \varphi_i) - c_j \sin(x + \varphi_j) \tag{3}$$

Using the same identity as above (3) expands to

$$c_i \cos(\varphi_i)\cos(x) + c_i \sin(\varphi_i)\sin(\mathrm{x}) - c_j \cos(\varphi_j)\cos(x) + c_j \sin(\varphi_j)\sin(\mathrm{x}) \tag{4}$$

Which simplifies to

$$d_1 \cos(\mathrm{x}) + d_2 \sin(\mathrm{x}), \tag{5}$$
$$d_1 = c_i \cos(\varphi_i) - c_j \cos(\varphi_j), d_2 = c_i \sin(\varphi_i) - c_j \sin(\varphi_j)$$

We can write (5) in the form

$$\mathrm{A}\sin(x + \alpha), \tag{6}$$

where,

$$\mathrm{A}\cos(\alpha) = d_1, \mathrm{A}\sin(\alpha) = d_2.$$

But, $\mathrm{A}^2 = \mathrm{A}^2\cos^2(\alpha) + \mathrm{A}^2\sin^2(\alpha) = {d_1}^2 + {d_2}^2$. So,

$$\mathrm{A} = \sqrt{{d_1}^2 + {d_2}^2},$$
$$\mathrm{A} = \sqrt{\left(c_i \cos(\varphi_i) - c_j \cos(\varphi_j)\right)^2 + \left(c_i \sin(\varphi_i) - c_j \sin(\varphi_j)\right)^2}\,, \tag{7}$$

$$A = \sqrt{c_i^2 + c_j^2 - 2c_ic_j(\cos(\varphi_1)\cos(\varphi_2) + \sin(\varphi_1)\sin(\varphi_2))}$$

The phase shift, α, can be calculated as

$$\alpha = tan^{-1}\left(\frac{d_2}{d_1}\right) \tag{8}$$

For the example in the previous figures, $c_1 = 1, c_2 = 0.9, c_3 = 0.75, \varphi_1 = 0, \varphi_2 = 0.1, \varphi_3 = 0.2$. Note, the phase shift is expressed in seconds, not radians.

$$\begin{gathered} A_{12} = 0.59, \alpha_{12} = 0.42 \\ A_{13} = 1.05, \alpha_{13} = 0.37 \\ A_{23} = 0.53, \alpha_{23} = 0.31 \end{gathered} \tag{9}$$

For the three FCM command outputs from Figure 7, the resulting mid-value select algorithm generated the dark blue. The disagreement between the mid-value select and the individual channels can be calculated by simple differencing. This is illustrated in Figure 8.

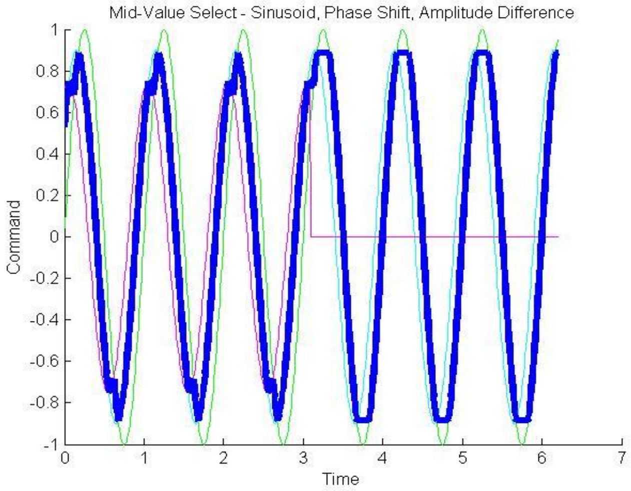


Figure 7 Mid-value selection from 3CMs, Channel 3 fails at time=3.0 sec.

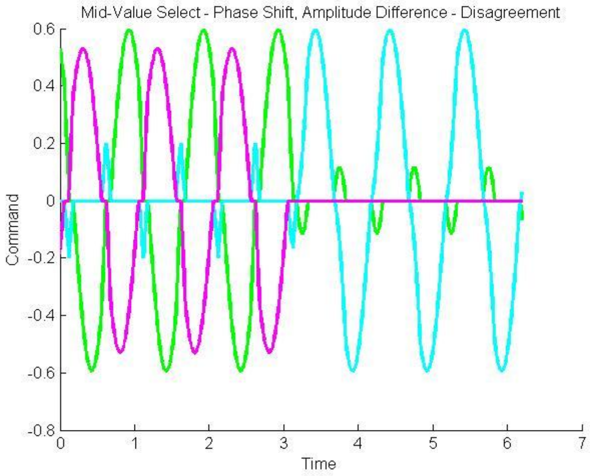


Figure 8 Disagreement between CM channels and Mid-Value Select.

5.1 Inconsistent Omission Error Force Fight

Figure 7 and the associated analysis allow us to calculate the force fight resulting from a dual Actuator Sense Module (ASM) with an inconsistent omission error with an asynchronous, sinusoidal plant. Illustrated in Figure 9, three signals are provided to one mid-value select algorithm, while only two signals are presented to the other ASM. The difference in the output of the MVS algorithms represents the force fight.

For the example presented in Figure 7, the period before the 3 second mark represents the output of ASM1 MVS. The period after 3 seconds represents the ASM 2 MVS. Note, in this that each MVS tracks a different signal for most of the time bounding the force fight by the sinusoid defined in (6). The force fight is presented in Figure 10.

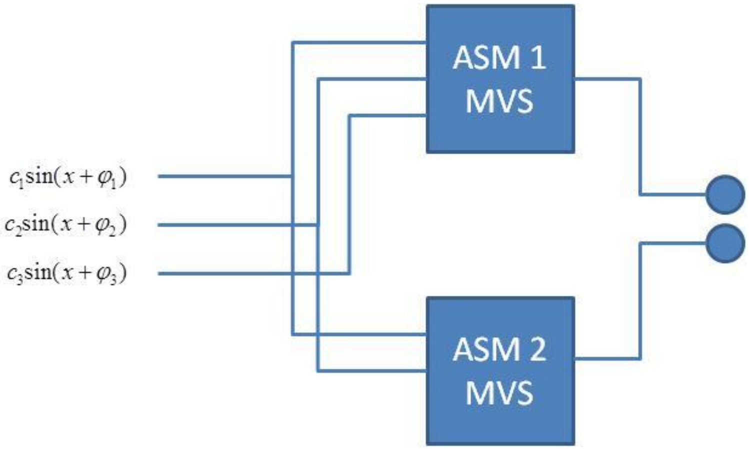


Figure 9 Dual ACE Force Fight - Inconsistent Omission Error.

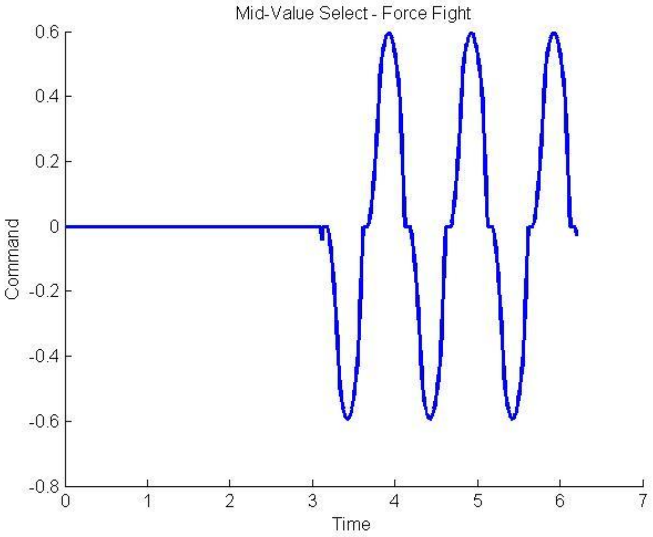


Figure 10 Force Fight - Inconsistent Omission Error at t=3.0 sec.

In the earlier SAL analysis a triangle wave command was used in lieu of a sin wave. Figures 11 presents the equal amplitude commands used in the SAL study. Figure 12 presents the force fight when either Channel 1, 2, or 3 fail.

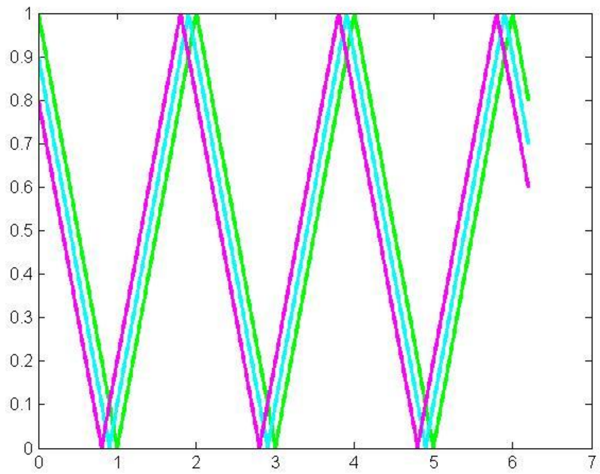
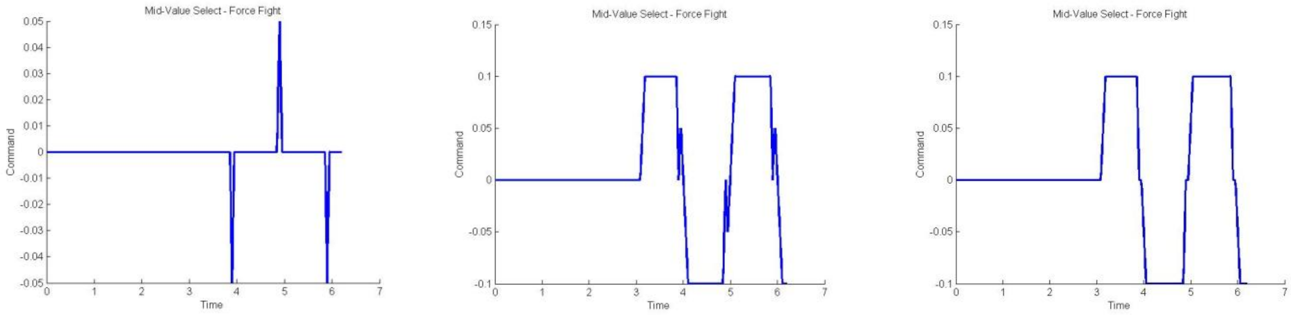


Figure 11 Triangle Wave used in SAL Analysis.



Channel 1 Fail

Channel 2 Fail

Channel 3 Fail

Figure 12 Force Fight from Triangle CM Commands - Single Channel Inconsistent Omissions.

6 Conclusions and Future Work

The Phase-2 case-study architectures [1] present a number of interesting challenges with regard to formal verification. Originally, it was our intention to form an integrated model that would support the verification of the agreement properties within the context of the plant model and control system logic. Our hope was that the SAL and Hybrid SAL tool chains would support the model capture and formal verification. However, the lessons learned from the open-loop simplified model indicate that such a strategy may not be the best option and that the SAL tool chain may not be the best vehicle to develop the proof. The effort and time required to manually develop the auxiliary lemmas to support the required proof was significant. Although automated invariant generation methods do exist, for example [15], these methods have not yet been implemented within SAL. We further conjecture that the invariant used by the mvs model may be beyond what these automated method can find because auxiliary variables are required. Given this experience at the time of writing we believe that it may be preferable to use SAL as a debug tool and use PVS or related technologies to formalize the proof arguments. In addition, the visual results using MATLAB allowed us to develop an intuition that was then used to form a closed form solution to the force fight for both sinusoidal and triangle CM command waveforms. When learning new technologies, such as SAL, it is easy to get lost in the intricacies of the new tool and language, which may allow more simple solutions to be overlooked.

Given the above, it is our intuition that an integrated model may not be the best approach and attacking the problem in stages may be preferable. For example, use SAL or Hybrid SAL to characterize and potentially prove the control feedback characteristics (e.g. maximum rate of change, etc.) then use these abstracted characteristics to formally investigate agreement in a separate module. At the current time, the ability to reason and verify complex agreement strategies that incorporate cross-lane equalization and mode consolidation within the control logic is also uncertain. This will be an area of focus in Year 3 of the work.

The work performed to date, focusing on the output agreement, has proved to be very educational with respect to our understanding of the distributed agreement properties under inconsistent omissive and Byzantine fault-scenarios. We believe that the alternative behavior and properties of the BRAIN 3.0 protocol is also an interesting contribution to such architectures, supporting a discussion of agreement without the asynchronous vs. synchronous system philosophical discussions.

In upcoming work, we intend to focus on the asynchronous architectures. We intend to refine the techniques to develop an integrated argument. We further intend to augment the model with representative lane-equalization, input selection, and asymmetric fault management logic. We also intend to explore the system validation actives performed on similar real-world system to assess the applicability and feasibility of applying the analysis developed here-in. We will also investigate the feasibility and potential benefit that can be derived from tests generated from the formal model abstractions.

We further intend to extending this work to investigate more elaborate equalization strategies such as [14], and to explore the issues of multi-rate system [15].

Finally, we intend to characterize these implications of the asynchronous control architecture and to contrast the bandwidth and CPU efficiency of our three case-study architectures; for example, to formally evaluate the required computational for equivalent levels of agreement.

References

[1] B. Hall, K. Driscoll, and K. Schweiker, “Verification and Validation of Flight Critical Systems Research Project: Report Detailing Phase-2 Case Study Development,” to be published.

[2] K. Driscoll, G. Madl, and B. Hall, “Modeling and Analysis of Mixed Synchronous/Asynchronous Systems,” NASA/CR-2012-217765, September 2012.

[3] S. Osder, "Chronological overview of past avionic flight control system reliability in military and commercial operations," AGARD-AG-224 , P. R. Kurzhals, ed., NATO Research and Technology Organization, vol. 224, Jan 1977, pp. 2-1-2-17. Available from NTIS HC A16/MF A01.

[4] S. Osder, "Generic Faults and Architecture Design Considerations in Flight Critical Systems," *AIAA Journal Of Guidance, Control and Dynamics*, vol. 6, no. 2, March-April 1983, pp. 65-71.

[5] B. Wittenmark, B. Bastian, and J. Nilsson, “Analysis of time delays in synchronous and asynchronous control loops,” *Decision and Control, 1998. Proceedings of the 37th IEEE Conference on*, vol. 1, pp. 283–288, 1998.

[6] V. A. Regenie, C. V. Chacon, and W. P. Lock, “Experience with synchronous and asynchronous digital control systems," NASA Ames Research Center, NASA Tech. Mem. 88271, August 1986. Presented at the AIAA Guidance, Navigation, and Control Conference, Williamsburg, Virginia, August 18-20, 1986.

[7] SAE AS6802 November 2011 Time-Triggered Ethernet.

[8] SAE AS6003 February 2011 TTP Communication Protocol.

[9] A. Tiwari and B. Dutertre, “Modeling and Analysis of Asynchronous Systems Using SAL and Hybrid SAL,” NASA/CR-2013-217960, February 2013.

[10] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, “Byzantine fault tolerance, from theory to reality,” *Knowledge-Based Intelligent Information and Engineering Systems*, S. Anderson et al. (Eds): SAFECOMP 2003, Computer Safety, Reliability, and Security Lecture Notes in Computer Science Volume 2788, pp. 235–248, 2003.

[11] G.Davis, “An analysis of redundancy management algorithms for asynchronous fault tolerant control systems,” NASA Ames Research Center, NASA Tech. Rep. TM-100 007, 1987.

[12] K. Driscoll, B. Hall, and K. Schweiker, “Application Agreement And Integration Services,” NASA/CR-2013-217963, February 2013.

[13] B. Dutertre, M. Sorea, “Timed systems in SAL,” SRI Technical Report, *SRI-SDL-04-03,* July 2004. http://www.csl.sri.com/users/bruno/publis/sri-sdl-04-03.pdf (accessed April 11, 2013).

[14] L. R. Tomlinson and R. E. Freeman, “Signal selection and fault detection apparatus and method,” U.S. Patent 5,710,776, January 20, 1998.

[15] T. W. Johnson, “A qualitative analysis of redundant asynchronous operation,” *Proceedings of the IEEE 1978 National Aerospace and Electronics Conference NAECON 78,* May 16-18, 1978, Dayton, OH, USA, 1978.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
01- 04 - 2013

2. REPORT TYPE
Contractor Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Investigating Actuation Force Fight with Asynchronous and Synchronous Redundancy Management Techniques

5a. CONTRACT NUMBER
NNL10AB32T

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Hall, Brendan; Driscoll, Kevin; Schweiker, Kevin; 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-2013-217984

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul S. Miner

14. ABSTRACT
Within distributed fault-tolerant systems the term force-fight is colloquially used to describe the level of command disagreement present at redundant actuation interfaces. This report details an investigation of force-fight using three distributed system case-study architectures. Each case study architecture is abstracted and formally modeled using the Symbolic Analysis Laboratory (SAL) tool chain from the Stanford Research Institute (SRI). We use the formal SAL models to produce k-induction based proofs of a bounded actuation agreement property. We also present a mathematically derived bound of redundant actuation agreement for sine-wave stimulus. The report documents our experiences and lessons learned developing the formal models and the associated proofs.

15. SUBJECT TERMS
Distributed systems; Fault tolerance; Force fight; Redundancy management

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	39	(443) 757-5802