NASA/CR–2012-217765

Modeling and Analysis of Mixed Synchronous/Asynchronous Systems

Kevin R. Driscoll, Gabor Madl, and Brendan Hall
Honeywell International, Inc., Golden Valley, Minnesota

September 2012

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–2012-217765

Modeling and Analysis of Mixed Synchronous/Asynchronous Systems

Kevin R. Driscoll, Gabor Madl, and Brendan Hall
Honeywell International, Inc., Golden Valley, Minnesota

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL10AB32T

September 2012

Available from:

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

Abstract

Practical safety-critical distributed systems must integrate safety critical and non-critical data in a common platform. Safety critical systems almost always consist of isochronous components that have synchronous or asynchronous interface with other components. Many of these systems also support a mix of synchronous and asynchronous interfaces. This report presents a study on the modeling and analysis of asynchronous, synchronous, and mixed synchronous/asynchronous systems. We build on the SAE Architecture Analysis and Design Language (AADL) to capture architectures for analysis. We present preliminary work targeted to capture mixed low- and high-criticality data, as well as real-time properties in a common Model of Computation (MoC). An abstract, but representative, test specimen system was created as the system to be modeled.

Contents

1 Introduction

The documented work was performed under NASA Task Order NNL10AB32T, Validation And Verification of Safety-Critical Integrated Distributed Systems – Area 2.

1.1 Scope

This document is intended to satisfy the requirements for Deliverable 5.1.12 under Task 4.1.3.1 of this Task Order. It accompanies and provides the documentation for Deliverable 5.1.13, which includes models in electronic form. This modeling is work in progress and future changes can be expected.

The description of Task 4.1.3.1 says:

> The contractor shall develop advanced modeling and analysis capabilities to address emerging trends in integrated distributed systems architectures. In particular, the contractor shall define and model example mixed synchronous/asynchronous IMA architectures and applications. The contractor shall establish non-interference between time-triggered and asynchronous modes of communication. The contractor shall model and analyze fallback control from time-triggered operation to asynchronous mode to allow graceful degradation using a common network infrastructure. The contractor shall perform dependability, performance, and interaction analysis on models of example systems.

1.2 Motivation and Modeling

Trends in Integrated Modular Avionics (IMA) show increasing integration of mixed criticality data. Practical, safety-critical systems typically manage critical and non-critical data within the same platform. An emerging trend is the use of mixed synchronous/asynchronous communication patterns that enable designers to categorize data flows to achieve the best utilization given dependability requirements.

Researchers and developers of formal method tools would like to use real-world examples to test their research ideas and tool developments; however, creating such examples just for testing is prohibitively expensive. What is needed is an inexpensive way to share existing real-world designs in a format that can be used by formal methods tools.

This document presents an approach for the modeling and analysis of mixed synchronous/asynchronous systems. We build on the Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) language to capture distributed, fault-tolerant architectures for analysis.

One of the extensions to AADL is the "Annex E: Error Model Annex", which is contained in the SAE Aerospace Standard AS5506/1 [1]. The purpose of this Annex is to define an AADL standard compliant extension to the AADL core language for the support of dependability and fault modeling.

1.3 Tools

The AADL Error Annex work described in this report is based on AADL v1, Open Source AADL Tool Environment (OSATE) v1.5.8, and Error Annex plug-in version 1.1.7. All are freely available at `http://www.aadl.info`. The AADL model figures were created using the Error Detection Isolation Containment Types (EDICT) tool suite, available at `http://www.wwtechnology.com`. EDICT is based on the AADL v2 language. Other tools used include:

- The Symbolic Analysis Laboratory (SAL) is a formal model checker tool developed by SRI International, available at `http://sal.csl.sri.com`.
- The Distributed Real-time Embedded Analysis Method (DREAM) is an open-source tool and approach for performance estimation and real-time verification through Discrete Event Simulations (DESs). DREAM can also automatically generate timed automata models for the UPPAAL (`www.uppaal.com`) and Verimag IF (`http://www-if.imag.fr`) model checkers.

2 Background

2.1 Timing Terminology

2.1.1 Standard Terminology

When discussing "synchronous" and "asynchronous" systems, it helps to have well-established definitions for these terms. A good set of definitions for timing terminology can be found in Recommendation G.701 of the International Telecommunication Union (ITU), titled "Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms."[2] We quote the relevant text below(non-English synonyms removed). Terms in square brackets are in common practice but their use is deprecated in the sense defined in the Recommendation.

> 6014 **isochronous**
> The essential characteristic of a time-scale or a signal such that the time intervals between consecutive significant instants either have the same duration or durations that are integral multiples of the shortest duration.
>
> NOTE – In practice, variations in the time intervals are constrained within specified limits.

> 6015 **anisochronous**
> The essential characteristic of a time-scale or a signal such that the time intervals between consecutive significant instants do not necessarily have the same duration or durations that are integral multiples of the shortest duration.

> 6016 **synchronous [mesochronous]**
> The essential characteristic of time-scales or signals such that their corresponding significant instants occur at precisely the same average rate.
>
> NOTE – The timing relationship between corresponding significant instants usually varies between specified limits.

> 6017 **homochronous**
> The essential characteristic of time-scales or signals such that their corresponding significant instants have a constant, but uncontrolled, time relationship with each other.

> 6018 **non-synchronous [asynchronous/heterochronous]**
> The essential characteristic of time-scales or signals such that their corresponding significant instants do not necessarily occur at the same average rate.

> 6019 **plesiochronous**
> The essential characteristic of time-scales or signals such that their corresponding significant instants occur at nominally the same rate, any variation in rate being constrained within specified limits.
>
> NOTES
> 1 Two signals having the same nominal digit rate, but not stemming from the same clock or homochronous clocks, are usually plesiochronous.
> 2 There is no limit to the time relationship between corresponding significant instants.

6020 **heterochronous**
The essential characteristic of time-scales or signals such that their corresponding significant instants occur at different nominal rates.

NOTES
1 Two signals having different nominal digit rates, and not stemming from the same clock or from homochronous clocks are usually heterochronous.
2 Terms 6014 to 6020 are based on the following Greek roots:
iso = equal
homo = same
plesio = near
hetero = different

Most scholarly work and standards use definitions that are substantially similar to those quoted above. When wording in documents of these types differs from the wording of G.701, it is usually to make the definition easier to understand for readers unfamiliar with timing terminology. The cost of this accommodation is that the new definition may be ambiguous, imprecise/partially accurate, or even inapplicable to some situations. For example, such a definition (when read literally) may require absolutely zero jitter or may not apply to isochronous time-scales in which events are not equally spaced.

2.1.2 Applicability to Avionics

Isochronous components: Major components of avionics and other embedded digital electronic systems tend to have isochronous time-scales. These scales can be defined either explicitly or implicitly. Explicitly designed isochronous time-scales are designed to run at specific rates that are controlled by time signals derived from a clocking device such as a crystal oscillator. Implicit isochronous behavior emerges from components that aren't specifically designed to run at any particular rate (or rates), but run at an isochronous rate because their design is based on a repetitive loop of operations (either a hardware state machine or software). The time it takes to do one loop of these operations creates the period for an isochronous timescale. Both explicit and implicit isochronous designs will exhibit some degree of jitter.

Explicitly designed isochronous components can, and often do, have subcomponents that execute at harmonic rates. That is, each subcomponent runs at a rate that is an integer multiple of any lower frequency rates used by the other subcomponents. Because the simplest of these integer multiples (in terms of implementation on digital hardware) is two, this is the ratio most often used. One of the most popular isochronous rate hierarchies is 10, 20, 40, and 80 Hz.

Some components may have some isochronous subcomponents that are not synchronous with other isochronous subcomponents. Other components may have some subcomponents that are isochronous and other subcomponents that are anisochronous and non-synchronous with other subcomponents.

Synchronous systems: Saying that a system is synchronous can mean that any signal traversing a path, from system input to system output, through several isochronous components crosses only synchronous boundaries between these isochronous components. Or, in a redundant system, it can mean the timing between the redundant components in a redundancy set are synchronous. Or, it can mean both. That is, there are two orthogonal dimensions to synchrony in a system.

The differences between these meanings/dimension of "synchronous" can be described with reference to Figure 1. The inputs to this example system are the Pilot Input and Sensor components. Signals produced by these components traverse the system in a processing "pipeline" using the following order:

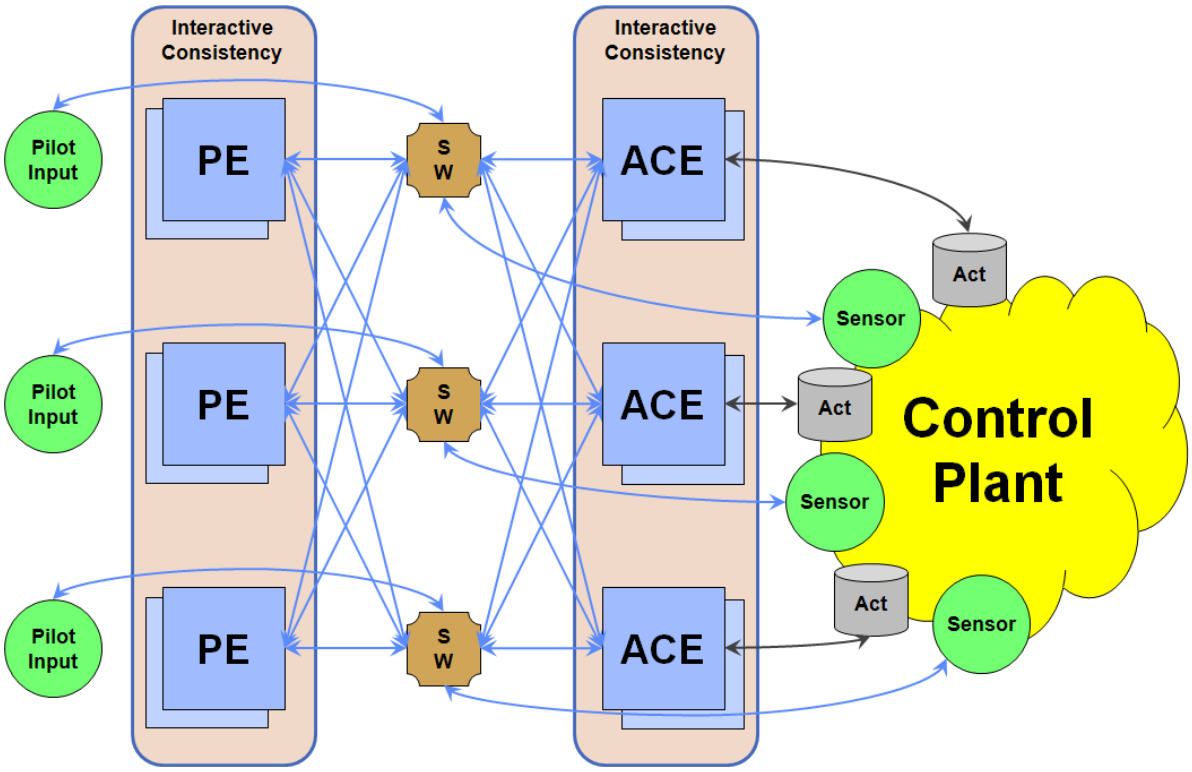

Figure 1. Example System

1. Switch (SW)
2. Processing Element (PE)
3. Analog Control Electronics (ACE)

In this figure, such signal flows can be seen as generally in the horizontal dimension.

The figure also shows redundancy. Members of a redundancy set are separated vertically. Thus, in the context of this figure, we can talk about "horizontal synchronization" for the input-to-output pipeline and "vertical synchronization" for the redundant elements. Any such pipelined and redundant system can be represented similarly; therefore, the concept of two dimensions of synchronization applies to all such systems.

Unless otherwise specified, when an avionics designer says a system is synchronous, this usually means that the system is synchronous in both dimensions. We use this convention in this document.

For any given system, one mechanism can provide both dimensions of synchronization or each dimension could be provided by a separate mechanism. In a design using separate components for each dimension, a component may have difficulties when the horizontal and vertical synchronization mechanisms try to adjust the time-scale in opposite directions. While this problem is rarely discussed in the literature, a system design must resolve this conflict such that requirements for both dimensions of synchronization are satisfied simultaneously.

Isochronous systems: The term "asynchronous" has been ambiguous since the very beginning of digital avionics system design. It is may be used to mean either anisochronous or non-synchronous or both, often without sufficient context to resolve the ambiguity. Per the G.701 recommendations,

the use of this term has been deprecated in favor of the unambiguous terms anisochronous or non-synchronous. However, because this term is so ingrained in digital avionics design and most digital avionics system designers are unfamiliar with the unambiguous terms, the (mis)use of this term is likely to continue for the foreseeable future. For this reason, we continue to use the term "asynchronous" where it may be expected by these designers and where its use is either unambiguous or its precise definition is unimportant. In other cases, the unambiguous terms are used. For this task, an "asynchronous system" means a system consisting of components that are each isochronous by itself and is not synchronous with any other component in either of the two dimensions discussed in Section 2.1.2.

Mixed systems: In the phrase "mixed synchronous/asynchronous IMA architectures" from this task's description (see Section 1.1), the term "mixed" may have either of two definitions. It could mean that some interfaces are asynchronous (only) and some are synchronous (only) (in this case, asynchronous means non-synchronous). Or, "mixed" could mean that some interfaces and/or components are both asynchronous and synchronous. For example, a component could have two isochronous subcomponents that are non-synchronous with each other, and this component interfaces with a similar component such that one subcomponent of each component is synchronous with the other via the interface while the other subcomponents are non-synchronous across that interface.

Timing adjustments: When correcting a component's time in order to maintain synchronization, the adjustments are rarely done at the crystal oscillator level. Some early work attempted correction at this level for fault tolerant clocks. But, because: (1) no analysis techniques have been developed that can establish a bound on the fault propagation effects for the analog techniques typically used, (2) the lowering cost of digital electronics has eliminated any cost advantage for these schemes, and (3) the difficulties in making such mechanisms work with the high-speed ($\sim$1 GHz) clocks used in today's processors, it has been many years since such mechanisms have been proposed. Instead, current techniques use some form of "sparse timebase" in which a high-speed clock is divided down to a lower speed clock via digital counters that can be dynamically adjusted to maintain the required rate and/or phasing of the lower speed clock.

2.2 The Synchronous/Asynchronous Debates

Debates over the relative merits of synchronous vs. asynchronous systems have taken place for decades. In these debates, these terms typically refer to systems in which the components are explicitly or implicitly isochronous and the discussion is whether these components should be synchronous to some "global" timeline or if each of the interfaces between the isochronous components should be asynchronous. In rare cases, the definition of asynchronous truly means event driven. However, most safety-critical systems are control systems that approximate continuous control rather than discrete event controllers.

A goal of this task is to shed some light on the synchronous vs. asynchronous debate and to provide useful comparison of their characteristics. In particular, we address some of the issues posed in Section 3.5.

3 Example System

3.1 Selection of Test Specimens

Test specimen selection or creation for formal methods research occurs along a trade-off spectrum. At one end of the spectrum, are problems that are so simplified that they bear little resemblance to reality, and the results of formal methods research using these specimens can be called into question. At the other end of the spectrum, one can design a full system exactly as it would be fielded. Such a system would have to be developed "from scratch" to avoid proprietary and ITAR issues (the results of this research must be available for open publication). Avoiding these issues makes the design of such a system actually more expensive than designing a real system, because proprietary legacy components cannot be used. The cost of developing such a system can easily run into tens of millions of dollars—just to develop the test specimen before the research can begin. In addition, the complexity of a full system can obscure the research results (hiding the forest with leaves). Thus, useful test specimens are found in the part of the spectrum that provides the most meaningful results at reasonable cost.

The most important component of a dependable embedded system is its data network. The data network provides the structure for a system's architecture and also typically *defines* the system's architecture. It also tends to be the component responsible for providing the highest level of fault containment—that is, it is the fault containment barrier of last resort. Thus, getting the data network correct is of utmost importance.

Because of its importance, the data network should be the first consideration when selecting or creating a test specimen. Ideally, such a network should be an open standard that is accessible to anyone and widely used in safety-critical systems. Sadly, networks that fully support mixed synchronous/asynchronous traffic have not yet entered service, though some provide some extremely limited support. These networks are typically synchronous networks that provide some fixed allocations of bandwidth that can be used for "asynchronous" traffic. They include the ARINC 659 SAFEbus [3], FlexRay [4], and TTCAN [5]. Of these, only ARINC 659 is a standard and has the fault tolerance to be used in safety-critical systems. However, the Medium Access Control (MAC) mechanism used to arbitrate within its "asynchronous" allocations (called Master/Shadow windows) requires the synchronous portion of the protocol be working in order for the "asynchronous" arbitration to succeed.

TTEthernet is the only network that purports to support mixed asynchronous/synchronous traffic, has the fault tolerance for safety-critical systems, and has attained standard status. (The SAE AS6802 Time-Triggered Ethernet Standard [6] was published on November 1, 2011).

A newer version of the Braided Ring Availability Integrity Network (BRAIN) [7, 8, 9] has the possibility of being a network that supports mixed asynchronous/synchronous traffic and has the fault tolerance for safety-critical systems; however, it is still under development.

3.2 Example System Description

3.2.1 The Example System's Structure and Requirements

Structure: Figure 1 shows the structure of an Example System we created; it is replicated as the base for Figures 2 through 8). This structure consists of (from left to right in the figure) pilot inputs, Processing Elements (PEs), TTEthernet switches (SW), Analog Control Electronics (ACE), sensors, and actuators (Act) working cooperatively to control a plant (e.g., an aircraft's attitude and speed). Most of the communication among the components occurs via TTEthernet links (blue lines) that connect all the components, except for the actuators, to TTEthernet switches. The only

traffic that does not travel via TTEthernet are the direct links (gray lines) that connect analog control electronics to actuators. Each ACE drives just one of a triplex set of actuators.

Each type of component is triplex. In addition, the processing elements, analog control electronics, and all of the TTEthernet components (the switches and the interfaces to all the devices that communicate via TTEthernet) are fail-silent dual components (either self-checking pairs or command/monitor redundancy). We use "fail silent" rather than "fail stop" because the whole component may not have stopped operating; only one or more of its output ports either produced no output or produced an output that is obviously incorrect (to any observer). A proper subset of this "fail silent" failure mode is the inconsistent omission failure mode. We use "fail stop" for cases where a component has completely failed on all of its output ports and is not producing any output.

The pilot input and actuators are not fail silent. Each actuator can be made inert by a signal from its associated ACE that causes the actuator to go into a mode where the actuator provides no force or resistance to the other actuators. This signal occurs when commanded by an ACE or when the two halves of the ACE pair disagree.

The PEs compute an envelope-protected and stability-augmented control of the plant as long as failures do not prevent them from doing so. If all PEs or synchronization fail, the ACEs assume fallback control. In the fallback mode, the ACEs run asynchronously and provide no envelope protection or stability augmentation.

The PEs' inputs come from the triplex pilot inputs and the triplex sensors. The PEs' outputs are sent to the ACEs, which then use the outputs to control the actuators. If all PEs or synchronization fail, the pilot inputs go to the ACEs instead of PEs and sensor input is not needed.

Requirements: The system needs to tolerate two uncorrelated faults with one possibly being Byzantine. The system also needs to minimize force fight between the actuators. A force fight occurs when multiple actuators try to move a flight surface to different positions, which results in the actuators exerting forces that, to some degree, oppose each other. The magnitude of the force fight is the difference in the forces applied by the actuators. To save power, the force fight should be as close to zero as possible. To prevent mechanical fatigue failure, the force fight must be less than 10% of the actuator input signal's full range. To help meet this requirement, the ACEs limit the rate of change in signals going to the actuators to be no more than 5% of the full-scale magnitude for each execution cycle of a thread. The total latency from pilot inputs and sensors to the actuators must be less than 50 ms. These requirements must be met in all modes of system operation.

This example system was chosen to have a diverse set of attributes found in real safety-critical systems, but simple enough to meet the specimen selection criteria given in Section 3.1 and to allow derivatives to be created easily (in order to test architectural variations, as described in Section 3.4). It contains the important attributes of many airplane flight control systems, but is not based on an actual flight control system.

3.2.2 An Example System Behavior

With the Example System structure and component descriptions from the previous Section 3.2.1, a number of different system behaviors can be created that satisfy the system requirements (also given in the previous section). This section describes one such set of behaviors.

Asynchronous and Synchronous Modes of Operation: The system has two modes of operation: asynchronous and synchronous. The asynchronous mode of operation is used as a minimal-

functionality fallback whenever the synchronization mechanism fails or all PEs fail. In the synchronous mode of operation, all components are synchronized to the TTEthernet timeline, which runs at a 20 Hz rate. In the asynchronous mode, all the components still run isochronously at 20 Hz, but none of the components are synchronized to any of the other components. Because of the increased latency and jitter when running asynchronously, the components run at 80 Hz when in the asynchronous mode.

Input Consistency: The fail-silent components (processing elements, analog control electronics, and all of the TTEthernet components) consist of a pair of subcomponents that operate such that each half of the pair does operations that are identical to the other half of the pair. The intention is that, given identical inputs to both halves of the pair, the identical processing will produce identical output (in the fault free case). This creates a derived requirement to ensure that the inputs to each of these pairs is identical between the two halves of the pair. This is done by having each pair exchange their inputs via a private communication path not shown in any of the figures for Section 3.

In addition to the two halves of each pair requiring identical inputs, each member of the pilot input redundancy set and each member of the ACEs redundancy set need either all of their inputs to be identical amongst the members of a set or the members of the set must exchange values to prevent internal state divergence (e.g. the value of integrators, mode state, etc.).

In the synchronous mode of operation, the PEs and ACEs try to ensure that their inputs are identical by exchanging a status frame[1] amongst themselves. For the consistency exchange usually seen in the literature about the Byzantine Generals problem, the entire contents of each input are exchanged amongst all of the recipients. However, systems that use fail-silent pairs (such as ARINC 659 SAFEbus and the TTEthernet used in our example system) can greatly reduce the required bandwidth by using "hierarchical Byzantine agreement". In this scheme, the two halves of a pair exchange the entire contents for each of their inputs, to make sure both halves have bit-for-bit identical inputs. This can be done rather cheaply because the two halves of a pair are typically adjacent to each other, which allows low-cost high-speed communication hardware to be used on a private link that does not contend for the communication resources of the inter-component network. For any input that is not identical, the pair rejects the input. For agreement among multiple pairs, the pairs only need to exchange one bit that says whether or not they accepted or rejected an input. In our example system, the PEs exchange one frame with six bits of payload – three bits to say whether or not it received each of the three pilot inputs OK or not, and another three bits to say whether or not it received each of the three sensor inputs OK or not.

If a PE fails to receive one of these status frames, it assumes that the status frame would have contained a status that matched its own. That is, any missing incoming status frame would have been identical to the status contents of the frame it, itself, had sent out. This covers the two most likely cases:
(1) The sending PE had the same status, but its status frame failed somewhere between the sender and receiver (the most probable case).
(2) The sender has failed stop (the second most probable case). For this case, it doesn't matter what the status would have been.
Similarly, the ACEs exchange status amongst themselves using six bits, three for the PE inputs and three for the pilot inputs. This scheme tolerates at least one Byzantine fault. It also tolerates an arbitrary number of Byzantine faults as long as no two faults have identical or supporting symptoms (colluding faults).

[1] TTEthernet, like all Ethernets, uses "frame" where most avionics designers could use the term "message"

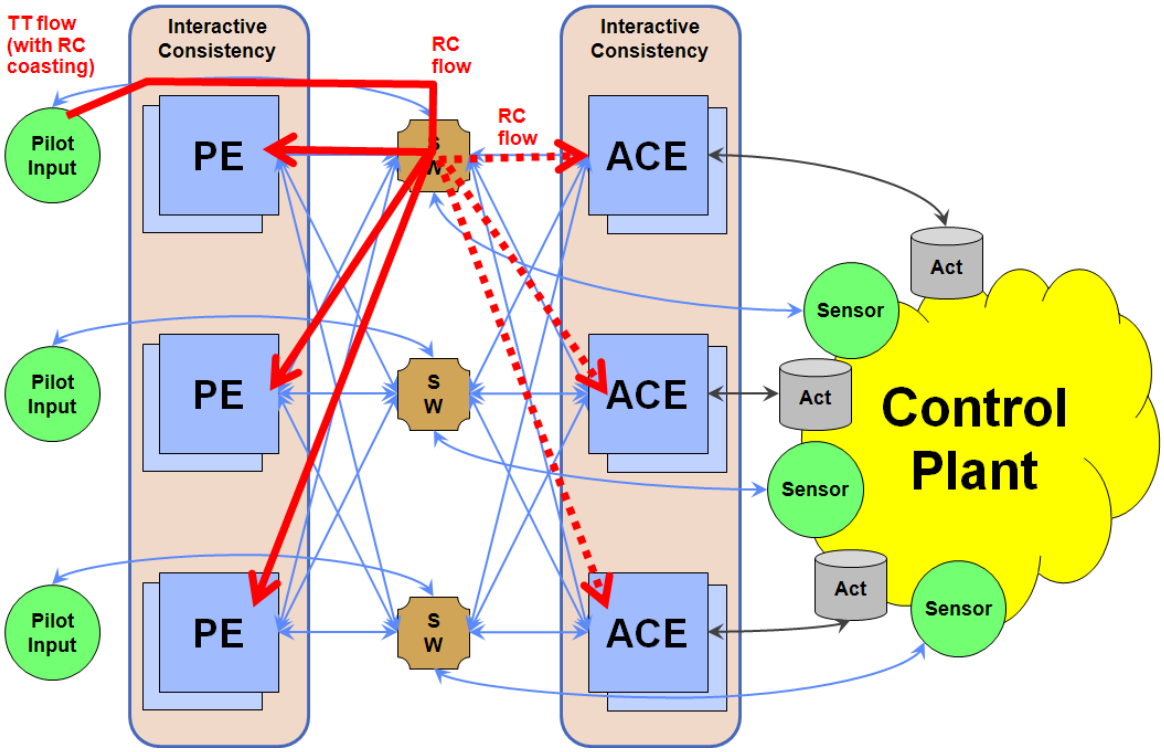

Figure 2. Pilot Input to PE Dataflow

Given no other traffic interferes (because there is no other traffic or it is small frames of low priority such that interference is negligible) and the fact the main players are isochronous means we have bounded communication delay (and Byzantine agreement is possible).

Because the ACEs have to operate in the asynchronous mode, whereas the PEs do not, the ACEs need to include some additional information in the exchanges amongst themselves. Without the aid of a synchronous timeline, each ACE cannot be certain which status bit belongs to which PE or pilot input frame, i.e. does a particular status belong to the $nth-1$, nth, or the $nth+1$ frame. So, each of these status bits must be accompanied with some form of sequence tag. Then, each ACE must implement some form of "waiting and matching section" to group status bits from the same frame together. In addition, the asynchrony among the ACEs means that their internal state can diverge even in the fault free case. This requires that the ACEs do state exchanges whenever they're running asynchronously.

Data Flows: The system's communication data flow, from inputs to output, can be scheduled in the synchronous case to occur in the following phases.

First, the pilot inputs are sent to the PEs. Figure 2 shows the top of input being sent to the three PEs. At the same time, the second and third pilot inputs are sent to all three PEs. Only the first pilot input is shown in the figure in order to make the figure not too confusing. In addition to these pilot inputs going to the PEs, the same data is sent to the ACEs. This latter transmission is not needed for synchronous mode (which is a reason for the dotted lines in the figure). It is used to simplify the transition from synchronous mode to asynchronous mode. The transmission from the pilot inputs to the PEs uses TTEthernet's time-triggered (TT) class of service. This class of service minimizes latency and jitter in a synchronous system by restricting how early and how late

a frame can arrive with respect to its scheduled transmission time. At the same time, the pilot inputs are routed to the ACEs using TTEthernet's rate-constrained (RC) class of service. This class of service uses a "leaky bucket" type of protocol to limit how often a frame can be sent (to prevent bandwidth hogging). TTEthernet's synchronization and timeline scheduling mechanisms do not need to be working in order for it to handle the RC class of service.

The dataflow from each sensor to the PEs is handled in the same way as for the pilot inputs, except that there is no RC flow to the ACEs. One of the three sensor flows is shown in Figure 3.

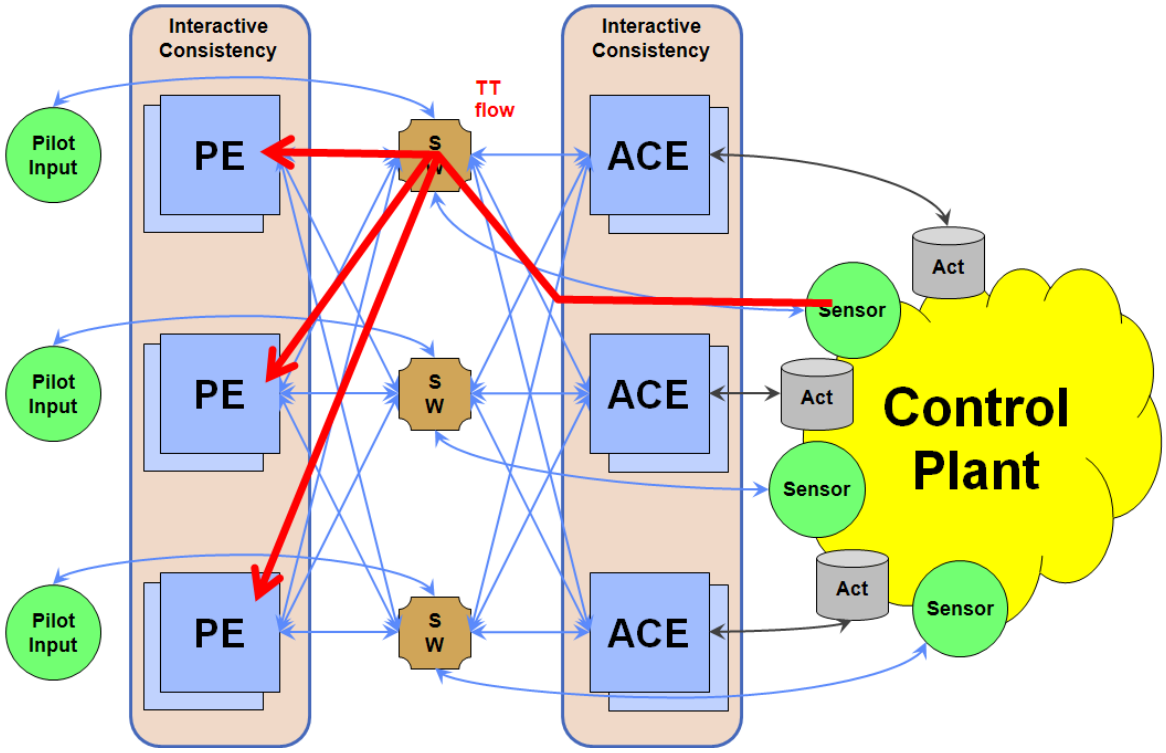

Figure 3. Sensor to PE Dataflow

After the PEs receive the sensor and pilot inputs, they do the status exchanges for these input frames. One example of this status exchange flow is shown in Figure 4. Because TTEthernet is a store-and-forward network, it takes one frame transit time to go from the PEs to the switches (all done in parallel). At the end of this time, all of the switches have all of the PEs' frames. Since each PE must receive a frame from the two other PEs, the transit time from the switches to the PEs must be two frame times. However, because these frames have only a few bits of status, or use the smallest Ethernet frame and the total transmission times is negligible.

The next data transfer phase is from the PEs to the ACEs. Again, it takes one frame time for all the PEs to get all of their frames to all of the switches. Then, the links from the switches to the ACEs must carry three frames (because each ACE must get input from all three PEs). This make the total propagation delay from the PEs to the ACEs equal to four frame times. The dataflow from one PE to the three ACEs is shown in Figure 5.

The next data transfer phase is for the input status exchange among the ACEs. While the dataflow patterns for this exchange are similar to that of the PEs, more data is exchanged because the frames that the ACEs exchange must carry the additional information needed for asynchronous operation. An example of the data flow from one ACE to the other two ACEs is shown in Figure 6. This transfer must use the RC traffic class because this dataflow must work for both the synchronous

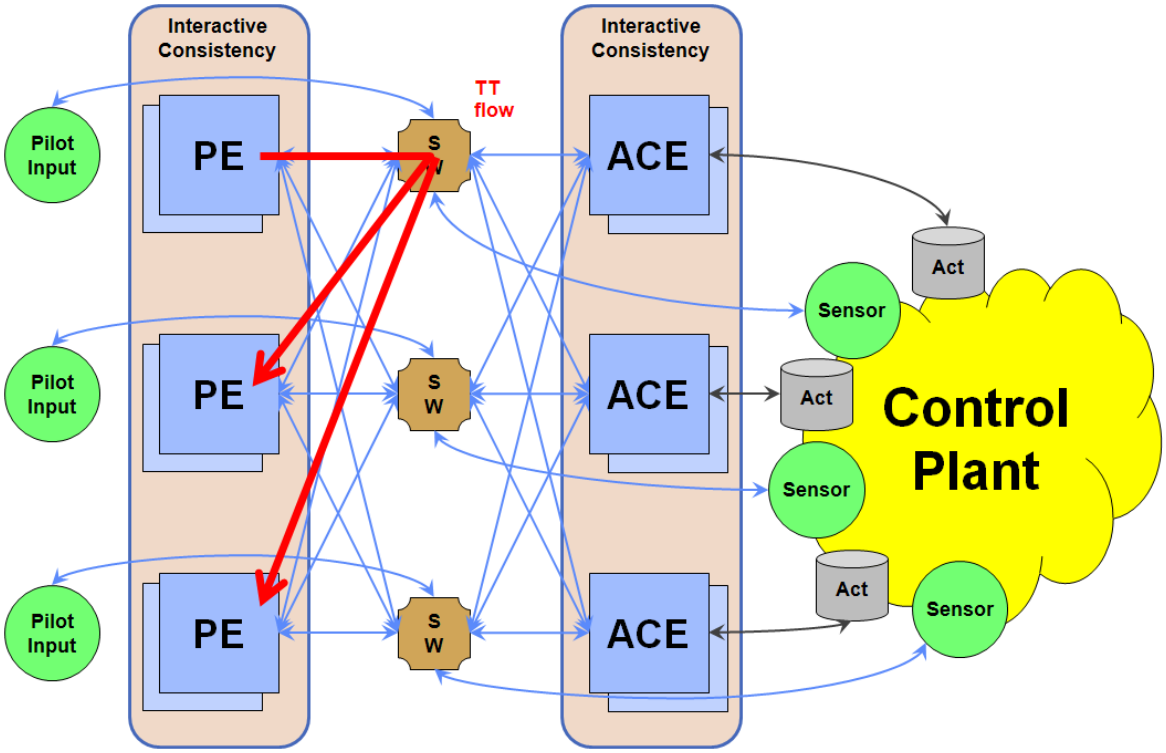

Figure 4. PE to PE Dataflow

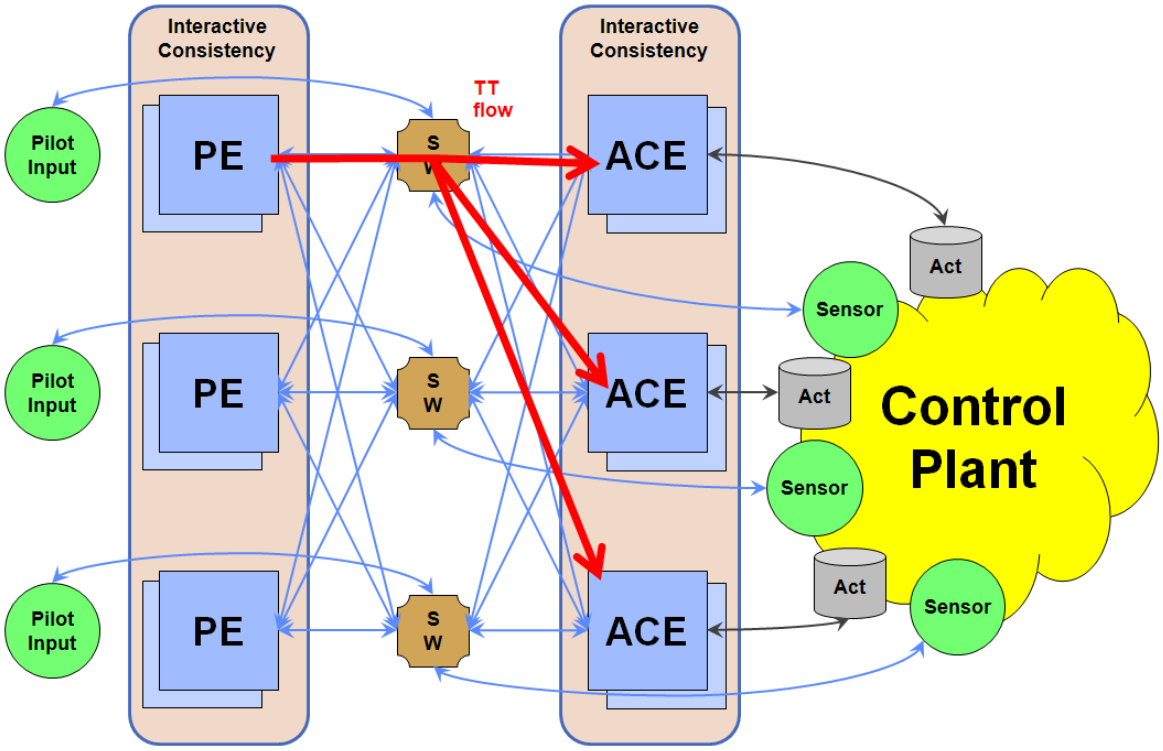

Figure 5. PE to ACE Dataflow

and asynchronous modes of operation.

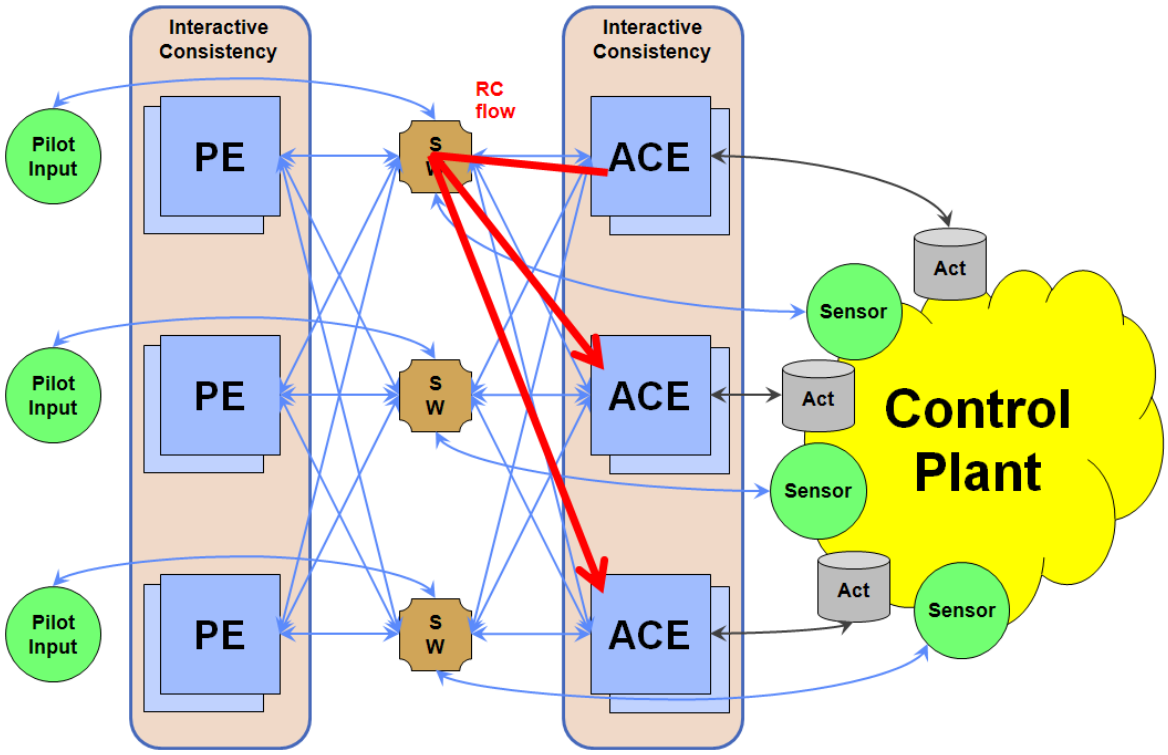

Figure 6. ACE to ACE Dataflow

The next communication phase is from the ACEs to their actuators. This communication is point-to-point from one ACE to one actuator, replicated three times. This communication involves no protocol, it is just direct wires. Figure 7 shows all three of these ACE to actuator flows.

The preceding paragraphs capture all of the communication flows for the synchronous mode of operation. The asynchronous mode of operation reuses the flows shown in Figure 6 and Figure 7. The other flows for the synchronous case are not used in asynchronous operation. There is an additional flow for asynchronous case, which is shown in Figure 8. This flow replaces the first flow described for the synchronous case (the description associated with Figure 2).

3.3 Synchronous Timelines

The timing relationships for TTEthernet communication are established by design-time schedulers. A simplistic timeline for this example system is shown in Figure 9. Much lower latency could be achieved by a more sophisticated scheduler. The simplistic timeline is shown here because it is easiest to understand.

3.4 Possible Derivative Systems

The Example System is easy to change to look at the implications of architectural variations. Some of the changes that could be made include the following.

- replace TTEthernet as the example communication infrastructure, e.g. with a BRAIN having store-and-forward mechanisms for mixed synchronous and asynchronous operation

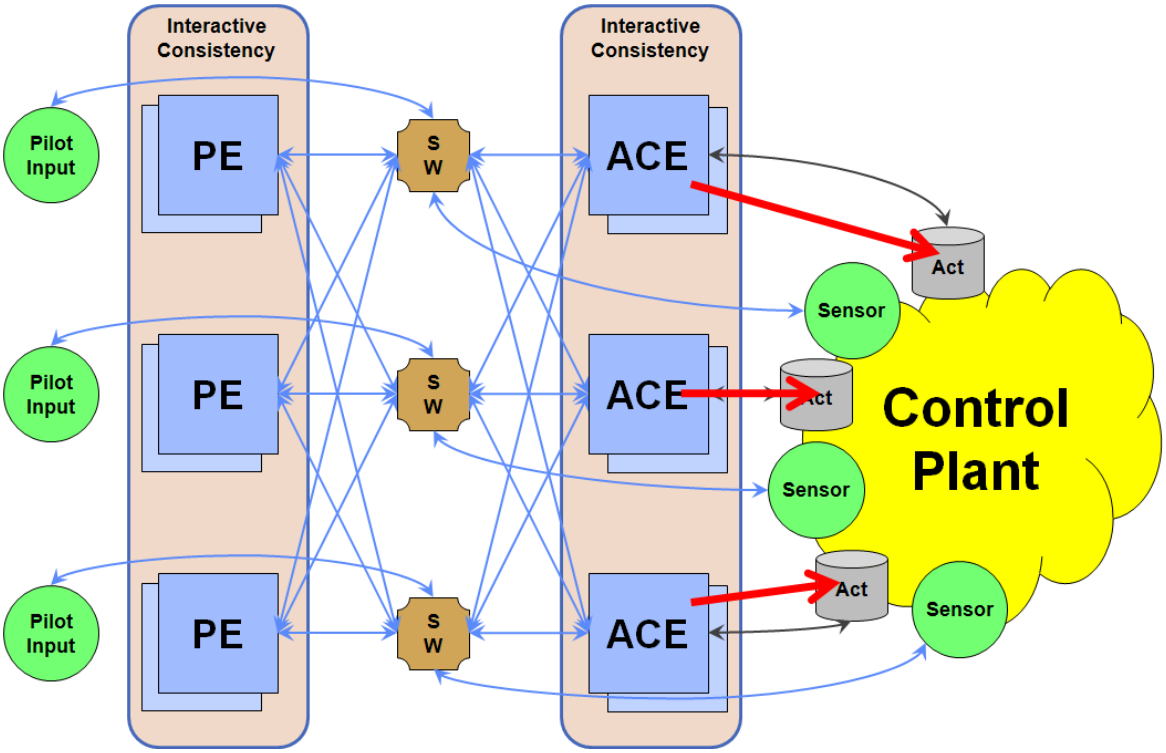

Figure 7. ACE to Actuator Dataflow

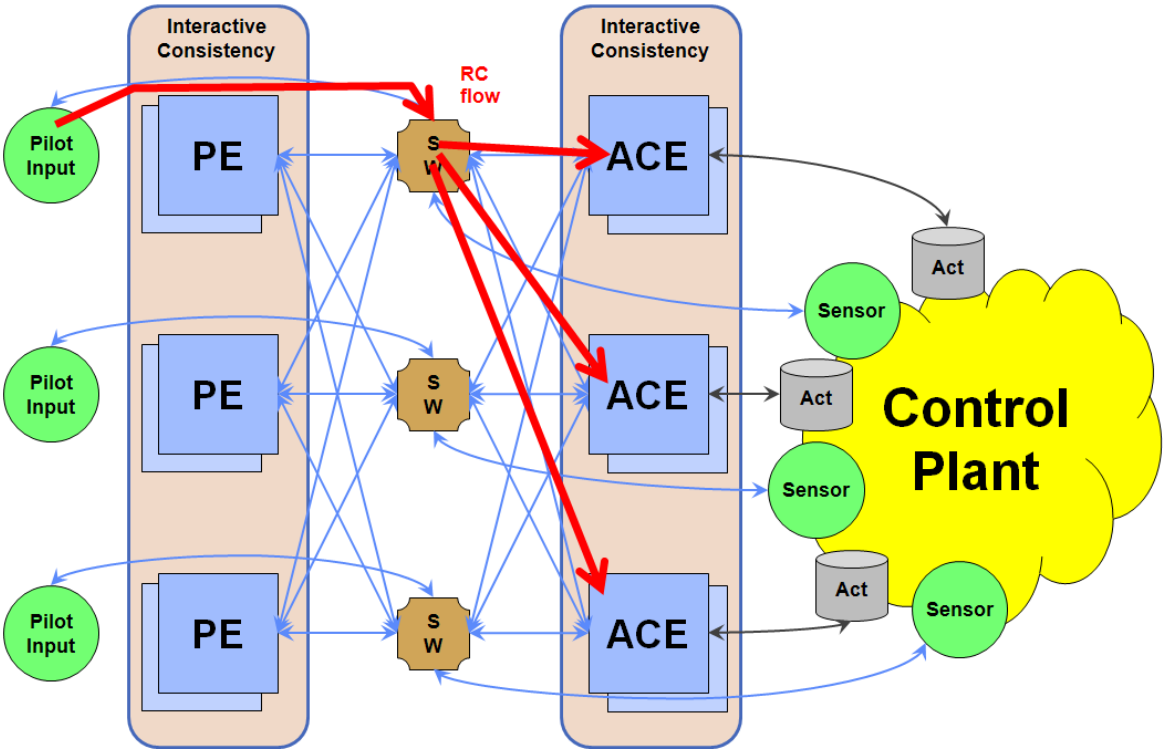

Figure 8. Pilot Input to ACE Backup Dataflow

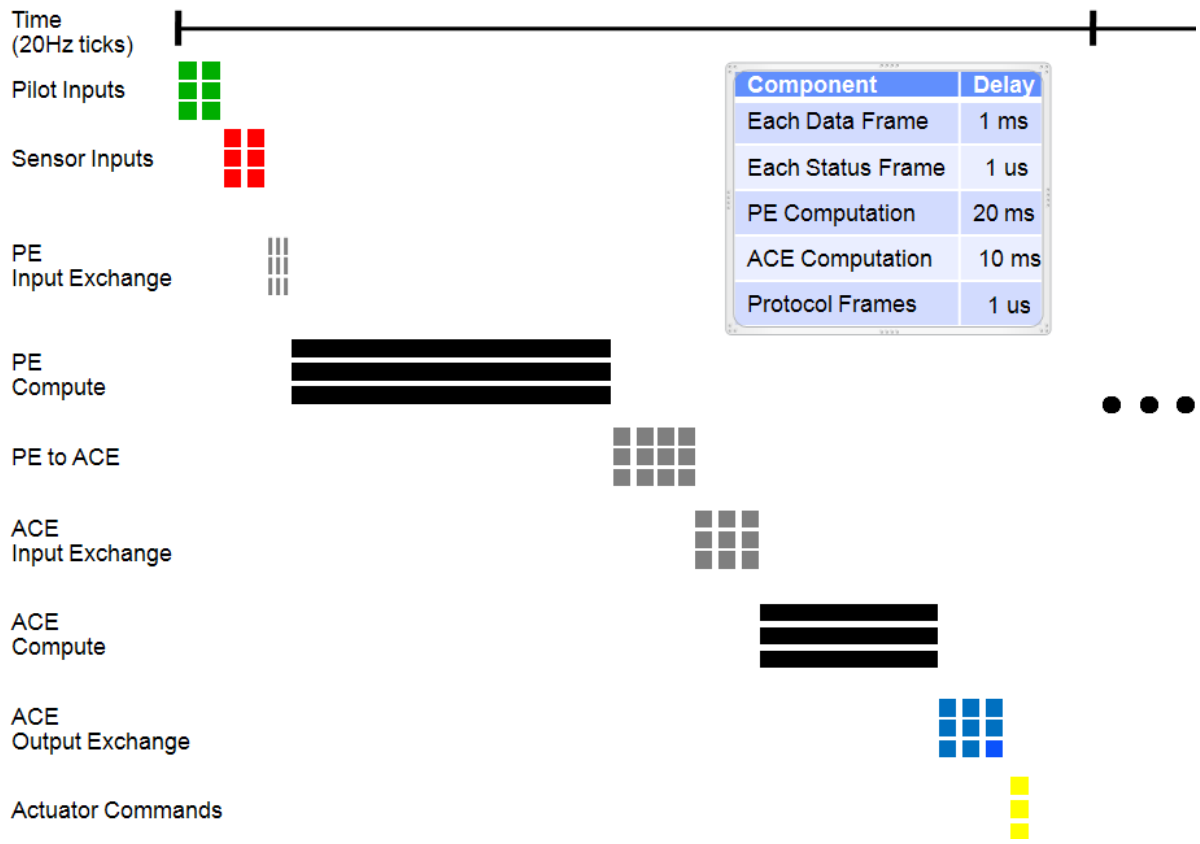

Figure 9. Simplistic Timeline

- make the system fully synchronous
- make the system fully asynchronous
- replace fail-silent components with simplex components
- try different methods for preventing state divergence and mode inconsistency
- the resource use of the pilot input threads non-negligible
- add mechanisms for recovery of failed components (e.g. transient fault recovery)

3.5 Questions to Be Answered

A system designer faced with making decisions about designing a system that may be asynchronous, synchronous, or both needs to answer a number of questions. Some of the questions that formal methods and modeling can help answer are presented in this section.

3.5.1 Resource Comparison

How do the resources required for asynchronous operation compare to synchronous operation?

In our example system, we will say that synchronous mode uses 20% of each PE's resources, 10% of each ACE's, and a negligible amount of the pilot input. Further, to simplify analysis, we will say that each frame consumes 1 ms of link bandwidth. Switch buffers are a little harder to

quantify. One way of doing this is to say that one unit of buffer is the space required to hold 1 ms of traffic from one port. Note that TTEthernet in this example system uses store-and-forward (versus cut-through).

For communication bandwidth, we will use the numbers given in the inset within Figure 9. The message times, in milliseconds or microseconds, are the times for messages to traverse a single link from the sender's buffer to the receiver's buffer.

Is our estimate correct that asynchronous operation needs to run at 80 Hz in order to meet the requirements of 50 ms end-to-end latency and no more than 10% force fight?

3.5.2 Synchronous-to-Asynchronous Reversion

Can we create an algorithm/mechanism that successfully reverts from synchronous operation to asynchronous operation under any failure scenario allowed in the system's fault hypothesis? Can we prove that such an algorithm/mechanism is correct? Can we show that the transition period from synchronous to asynchronous causes no operation which violates any of the systems requirements?

3.5.3 Noninterference

Do any of the components or subcomponents that are asynchronous interfere with any of the components or subcomponents that are synchronous, or vice versa? Such interference comes from resource contention. In this example system, the possible resources that could be in contention are processor resources (e.g. CPU time, memory, etc.), link bandwidth, and switch buffer space. Processor resources exist in the pilot input components, the PEs, and the ACEs.

The pilot input components include two threads. One thread is synchronous to the TTEthernet timeline. The other thread runs at the same isochronous rate as the first thread, but is not synchronized to TTEthernet or to the first thread, making the pilot input an example of a mixed asynchronous/synchronous component.

The ACEs have two threads – one synchronous with TTEthernet and the other not synchronous with TTEthernet or the other thread. The difference between this threading model and the threading model for the pilot input components is that the two ACEs threads aren't active at the same time.

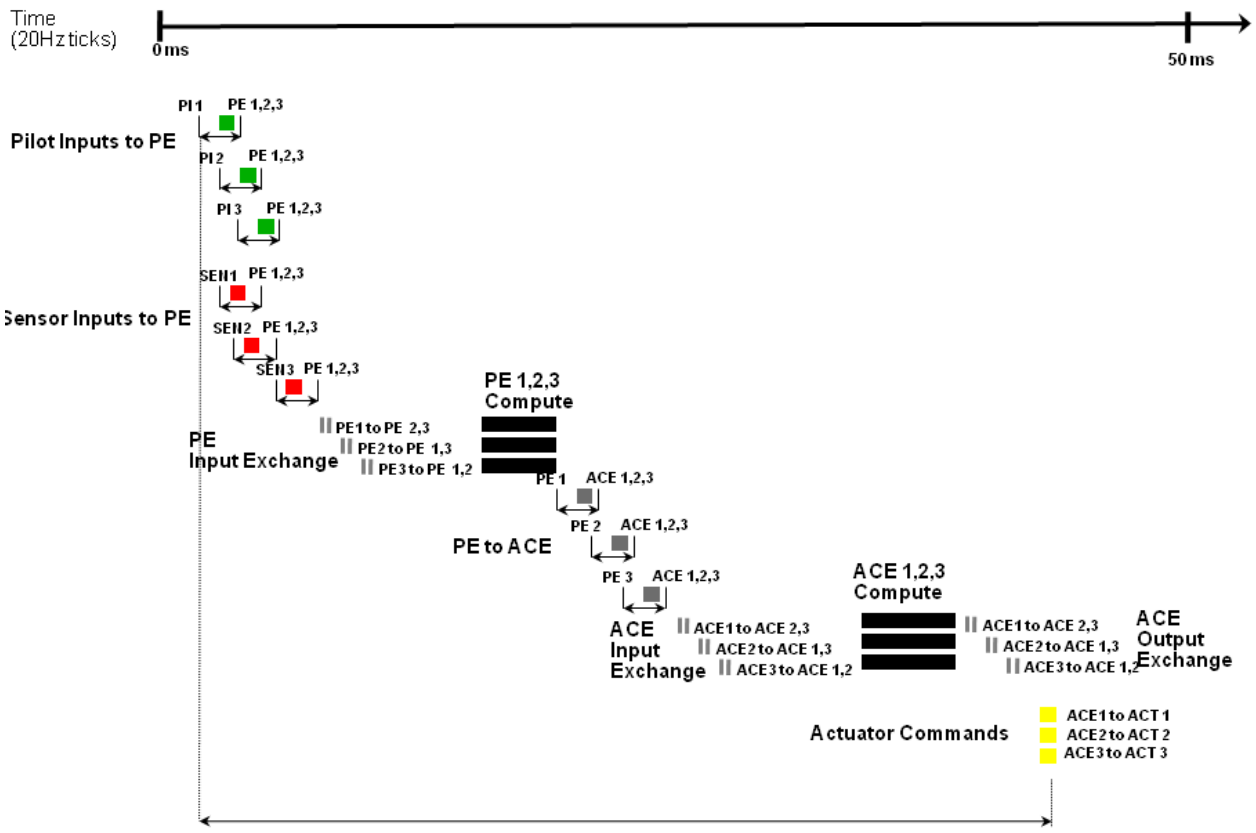

Figure 10. Sequence of Steps on a Timeline

4 AADL Modeling

This section describes how we applied Society of Automotive Engineers (SAE) Architecture Analysis & Design Language (AADL) to model the representative case studies described in Section 3.

4.1 TTEthernet Case Study: Synchronous and Asynchronous Models

The sequence of computation steps, data exchanges and additional exchanges for maintaining input consistency of data (described in Section 3) that need to be modeled on a timeline is shown in Figure 10. It illustrates the data flows between and the computations at each of the different nodes in the system, namely Pilot Inputs (PI), Processing Elements (PEs), Analog Control Electronics (ACEs), Sensors and Actuators. **The essential high-level requirement is that the closed loop system operates at 10Hz (100 ms period)**, i.e. round trip of 100 ms for Sensor to PE to ACE to Actuator and then back through the plant (actuator and aircraft dynamics, plus sensing lag) with at most a 50 ms budget for each direction.

Our objective is to study the resource utilization and latency of this system with (i) data flows modeled as time-triggered (TT) traffic representing a Synchronous system which utilizes a global time base in the system vs (ii) data flows modeled as Rate-Constrained (RC) traffic representing an Asynchronous system which does not rely on a time base and thereby cannot align phase relationships between the different nodes on the system and hence have to account for worst case behavior. Note that the figure is not drawn to scale or representative of actual times for SYNC

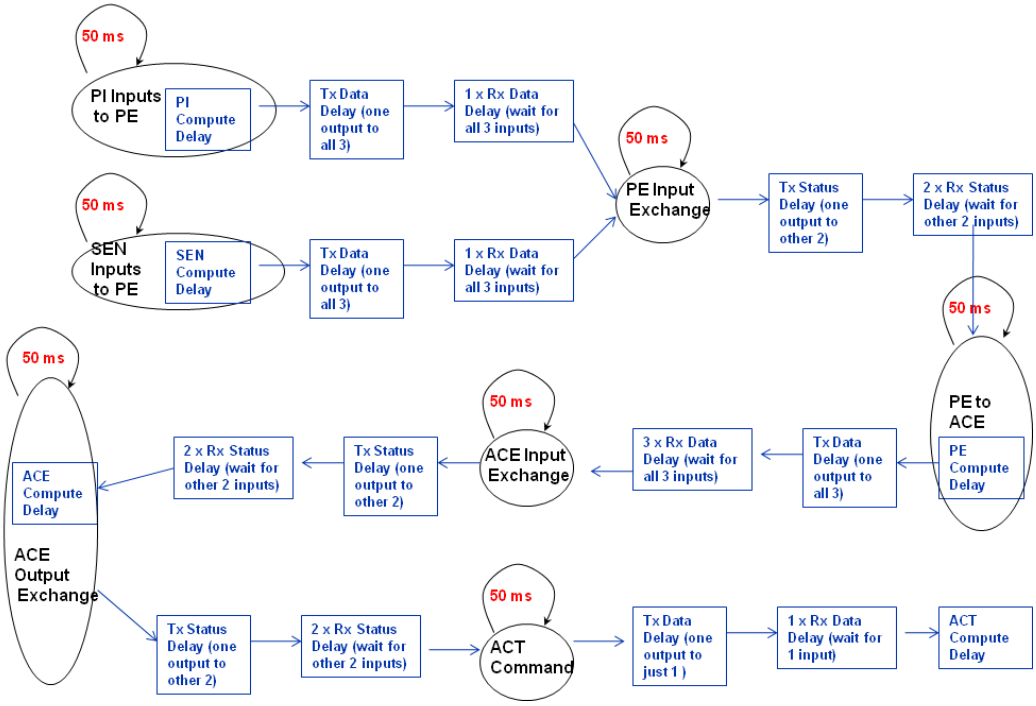

Figure 11. Delay Components End-to-End for Worst Case Synchronous (TT) Model

(TT) vs ASYNC (RC) cases. It has been staggered just for illustration purposes.

Figure 11 shows the state chart capturing the individual components of the delay that contribute to the end-to-end latency for the synchronous (TT) model of the example system. The computation delay at every node in the system is represented by the parameters *PI Compute Delay*, *SEN(sor) Compute Delay*, *PE Compute Delay*, *ACE Compute Delay* and *ACT(uator) Compute Delay*. The Inputs, Outputs and Computations at PI, SEN, PE, ACE and ACT all are modeled as periodic events at 50 ms period. Additionally, for maintaining data consistency, PEs and ACEs exchange data/status amongst themselves to maintain state congruence between themselves i.e amongst the 3 PEs and 3 ACEs. PEs have an Input Exchange with the other 2 PEs for maintaining consistency of data arriving from SEN and PI. Likewise, ACEs have both Input exchange with other two ACEs for data arriving from PEs and Output exchange with other two ACEs before sending data to ACT.

Finally the parameters *Tx Data Delay* and *Rx Data Delay* in Figure 11 represent the fixed (constant) network delay through the network for transmission and reception respectively for transferring data message flowing from source to destination. Note that the synchronous model assumes the network interface cards (NIC) and the nodes (hosts) are in phase, i.e. the network time base end-to-end, including clocks at NICs in the individual nodes, clocks in the underlying network switches and the clocks in the host processor in the individual nodes are all aligned with each other. Hence the time-triggered (TT) traffic flow requires low/zero jitter to transfer from Source host, through the Source NIC and underlying network through different switches, all the way till the Receiver NIC and the Receiver Host. Since TT traffic is always completely scheduled through the network whereby traffic interference is explicitly mitigated out through scheduling, data delays are represented as "fixed" constant delays in the synchronous model, with no variability, which can then be calculated a priori and plugged into the model. Similarly *Tx Status Delay* and *Rx Status Delay* represent constant delays for "status" messages, which may be smaller message size compared to data messages, which are primarily used for PE Input Exchanges and ACE Input and Output

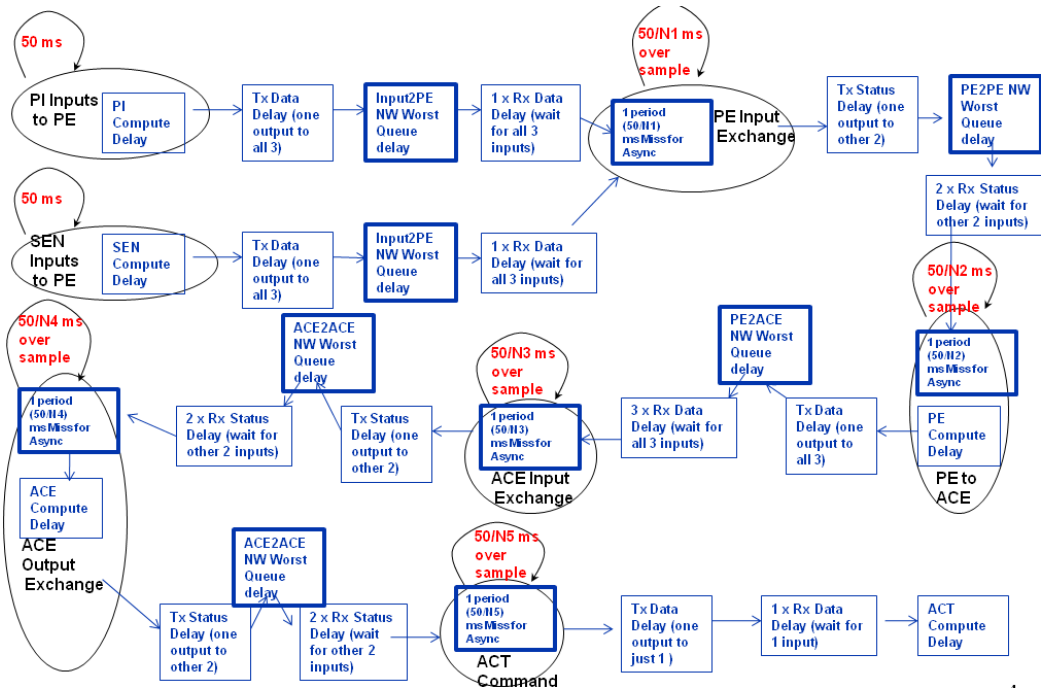

Figure 12. Delay Components End-to-End for Worst Case Asynchronous (RC) Model

Exchanges to exchange the vector of valid messages received in order to maintain consistency.

Similarly, Figure 12 shows the state chart capturing the individual components of the delay that contribute to the end-to-end latency for the asynchronous (RC) model of the example system. Comparing Figures 11 and 12, it can be observed that all the states and component delays in synchronous model are also present in the asynchronous model. There are two key differences which are described next.

Firstly, in the asynchronous model, the delays through the network are no longer fixed, constant delays. They are inherently variable latencies (though deterministically bounded) with non-zero jitter. This is primarily because the RC traffic is never scheduled anywhere in the network. Therefore, the worst-case interference of equal or higher priority traffic must be taken into account while calculating the delays. Since the complete set of dataflows in the system or workload is well defined and known a priori, the worst case network queuing delay is typically calculated using techniques such as network calculus or other approaches. *AADL v1 currently provides latency plug-ins which helps specify the delay components for both globally completely synchronous and asynchronous systems. But, AADL does not by itself provide the necessary queuing analysis techniques for worst case latency calculations and hence these need to be pre-specified in the AADL v1 model.* Hence, in order to model asynchronous systems, we introduce model parameters: *Input2PE NW Worst Queue delay*, *PE2PE NW Worst Queue delay*, *PE2ACE NW Worst Queue delay* and *ACE2ACE NW Worst Queue delay*. Each of these parameters represent the "worst-case" queuing delay for the RC dataflows PI to PE, PE to PE, PE to ACE and ACE to ACE respectively. In our model, we specify fixed, constant values for these parameters. The key assumption is that the system is analyzed through an off-line queuing analysis tool, upfront, and then the analysis results are applied to the AADL model.

Secondly, in AADL v1 (as far as we know), the latency analysis plug-in accounts for the "asynchronous" phase relation between the network bus and the tasks that run on processors in the model

Model Parameters	Constant Value (ms)
DataDelay (Tx Data Delay, Rx Data Delay)	1.0
StatusDelay (Tx Status Delay, Rx Status Delay)	0.001
$PIComputeDelay$	0.001
$SENComputeDelay$	0.001
$PEComputeDelay$	10.0
$ACEComputeDelay$	5.0
$ACTComputeDelay$	0.001
$Input2PENWWorstQueueDelay$	1.002
$PE2PENWWorstQueueDelay$	2.001
$PE2ACENWWorstQueueDelay$	4.0
$ACE2ACENWWorstQueueDelay$	5.0
$Period$	50.0

Table 1. Model Parameter Constants

by adding or penalizing a worst case one period to the end-to-end delay calculation. This is done to ensure all the communications on the network bus meet timeliness guarantees end-to-end because tasks on the processors are associated with local timeliness in an asynchronous mode instead of a global timeline and hence activities occurring in the same period boundary are not guaranteed. The net effect is that at every boundary (Host processor to Network NIC, Network NIC to Host processor) where there is potentially an asynchronous crossing of time bases, one period delay must be added. We also understand that there may be a more finer granularity mechanisms specifications for phasing relationships in the AADL v2 model (and the latency plug-ins) for asynchronous systems. At the time of this report in Phase I, we have not investigated AADL v2 and so are limited to AADL v1 asynchronous mechanisms provided.

Due to the above limitation in AADL v1, we are limited to adding one period to every asynchronous boundary crossing. The different asynchronous boundaries in our systems are at the different receiver at every one of the 5 stages, namely: *PE Input Exchange stage* (when data is received from PIs and/or SENs), *PE to ACE* (when data is received from PEs at the ACE), *ACE Input Exchange*, *ACE Output Exchange* and *ACE to ACT* (sending command to actuator). Adding one period (50 ms) at each of these boundaries would make the end-to-end closed loop infeasible because of our requirement that system needs to meet one-way latency of 50 ms or round-trip latency of 100 ms. In order to make asynchronous model feasible, we *over-sample* at each of these stages with parameters:$N1$, $N2$, $N3$, $N4$ and $N5$ where each $Ni \in 1, 2, 3..,$. As shown in Figure 12, this oversampling approach implies the periods are shortened at the corresponding receivers by $\frac{50}{N1}$, $\frac{50}{N2}$, $\frac{50}{N3}$, $\frac{50}{N4}$ and $\frac{50}{N5}$ thereby for some Ni setting the asynchronous models will results in a feasible solution of meeting one way latency of $\leq$ 50 ms. Note that oversampling results in *increased processor utilization (processor overhead) and network bandwidth reservation (bandwidth overhead).*

4.2 Constants for Model Parameters and Analytical Formulation for Worst Case Latency

The constant values used for the different model parameters discussed in previous Section 4.1 are listed in Table 1. These values were chosen so as to be reasonably representative of the actual system. The worst case queue delays given by *Input2PE NW Worst Queue delay*, *PE2PE NW*

Worst Queue delay, *PE2ACE NW Worst Queue delay* and *ACE2ACE NW Worst Queue delay* were appropriately calculated for the different traffic seen by the underlying network and taking a worst case arrival of those traffic as seen by the network.

Based on the state diagrams listed in Figures 11, 12, the worst case latencies can be calculated explicitly in a straight forward manner and is listed in Equations 1, 2 below respectively.

$$\begin{aligned} Latency^{sync}_{worstcase} =& Max(PIComputeDelay + 2 * DataDelay, SENComputeDelay + 2 * DataDelay) \\ &+ 3 * StatusDelay + PEComputeDelay + 4 * DataDelay + 3 * StatusDelay \\ &+ ACEComputeDelay + 3 * StatusDelay + 2 * Datadelay + ACTComputeDelay \\ =& Max(PIComputeDelay, SENComputeDelay) + PEComputeDelay \\ &+ ACEComputeDelay + ACTComputeDelay + 9 * Statusdelay + 8 * DataDelay \end{aligned} \tag{1}$$

$$\begin{aligned} Latency^{async}_{worstcase} =& Max(PIComputeDelay, SENComputeDelay) + PEComputeDelay \\ &+ ACEComputeDelay + ACTComputeDelay + 9 * Statusdelay + 8 * DataDelay \\ &+ Input2PENWWorstQueueDelay + PE2PENWWorstQueueDelay \\ &+ PE2ACENWWorstQueueDelay + 2 * ACE2ACENWWorstQueueDelay \\ &+ \frac{Period}{N1} + \frac{Period}{N2} + \frac{Period}{N3} + \frac{Period}{N4} + \frac{Period}{N5} \end{aligned} \tag{2}$$

Plugging the model constants listed in Table 1 into Equations 1, 2 , we get the Worst case End-to-End Latency/Delay shown in Table 2. While Synchronous model using TT traffic meets latency requirement of 50 ms, Asynchronous model using RC traffic must be **oversampled 26 times** in order to meet the 50 ms latency requirement.

4.3 Instantiating TTEthernet Case Study in AADL

The TTEthernet case study is a voted high-integrity architecture; it uses triple redundancy for all functional components.

Figure 17 shows the high-level overview of the TTEthernet-based case study. It consists of 3 `Pilot Input`, `PE`, `Switch`, `ACE`, `Sensor`, and `Actuator` components. We describe these components in more detail below.

Pilot Input: We have captured the `Pilot Input` component as an AADL system, as shown in Figure 13. The `Pilot Input` component consists of a Network Interface Controller (NIC), a processor, and a memory all connected through an on-chip interconnect. The processor executes a single thread implementing the `Pilot Input` functionality in software.

The `Pilot Input` component is a self-checking pair. Both elements in the pair have the same configuration, therefore we have omitted the second element of the pair from Figure 17.

Processing Element: The `PE` component, illustrated in Figure 14, performs computation based on the `Pilot Input` and `Sensor` readings to derive the optimal input values for the `ACEs`. The `PE` consists of a NIC, a processor, and a memory all connected through an on-chip interconnect, and a single thread running on the processor, executing the `PE` functionality in software.

Scenario						End-to-End Latency (ms) (Need to be $\leq 50ms$ in order to be feasible)
Synchronous Model						23.011
Asynchronous Model (oversampling period $50ms$ with $N1$,$N2$,$N3$,$N4$,$N5$ below)						
	$N1$	$N2$	$N3$	$N4$	$N5$	
	1	1	1	1	1	290.014
	2	2	2	2	2	165.014
	3	3	3	3	3	123.3473333
	10	10	10	10	10	65.014
	15	15	15	15	15	56.68066667
	20	20	20	20	20	52.514
	25	25	25	25	25	50.014
	26	26	26	26	26	49.62938462

Table 2. Theoretical Results from Analytical Computation

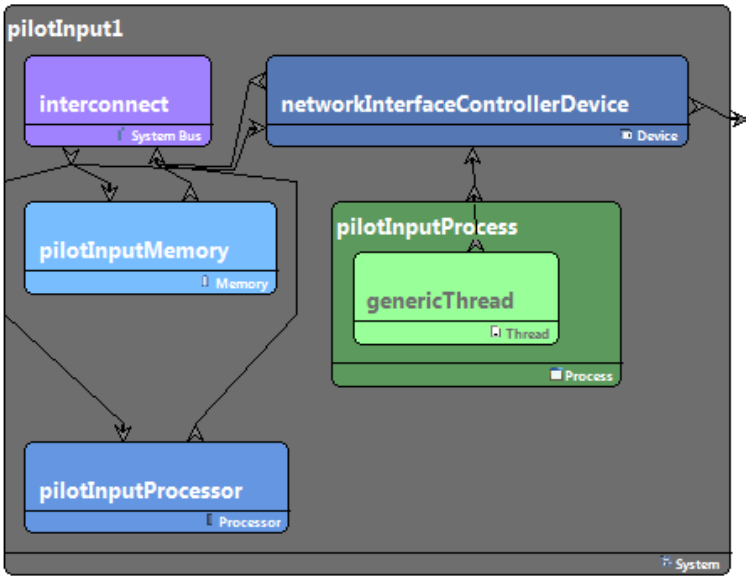

Figure 13. Pilot Input Component

Switch: The `Switch` component models the TTEthernet switch, managing communication over the network. We captured the switch as an AADL device, with multiple data ports corresponding to connections to NICs connected to the switch.

Analog Control Electronics: The `ACE` component models, illustrated in Figure 15, the electronics in charge of driving the `Actuators` based on signals received from the `PEs`. The `ACE` consists of a NIC, a processor, and a memory all connected through an on-chip interconnect, and a single thread running on the processor, executing the `ACE` functionality in software.

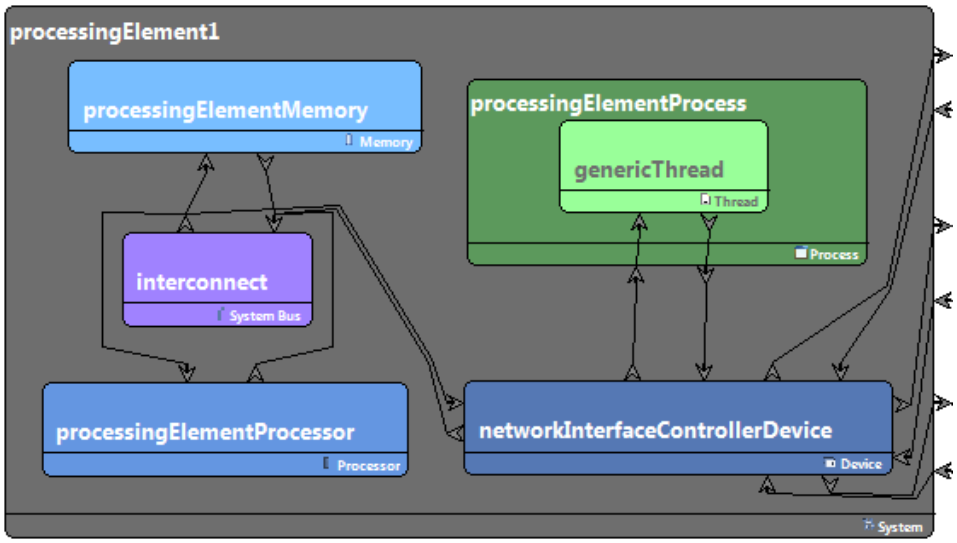

Figure 14. Processing Element Component

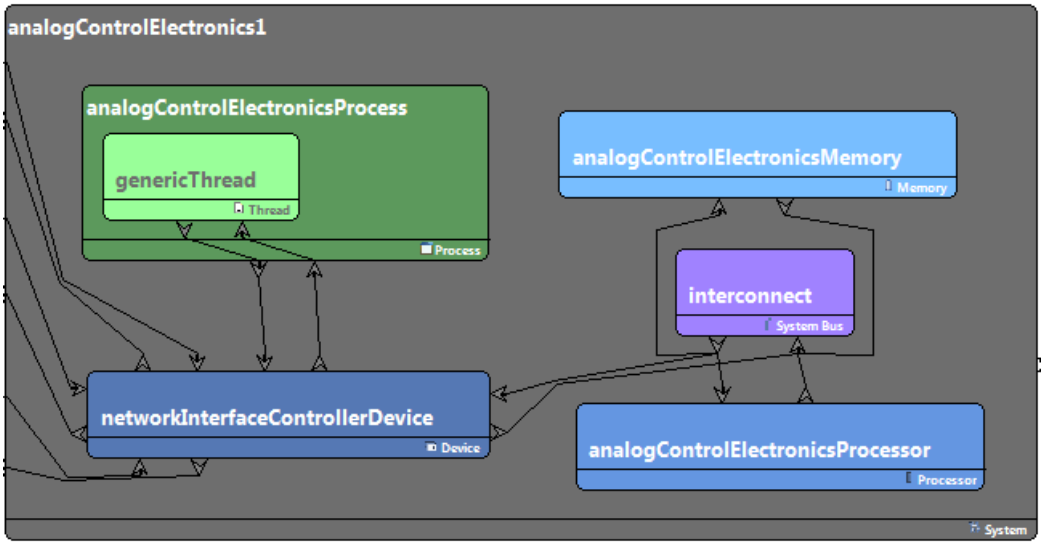

Figure 15. Analog Control Electronics Component

Sensor: The sensors provide data that is used to close the control feedback loop. Sensor data is fed back to the PEs, where it is used to augment the `Pilot Input` to the `ACEs`. We have modeled the `Sensor` as an AADL system consisting of a sensor device, and a NIC. The sensor AADL model is presented in Figure 16.

Actuator: The actuators were captured as AADL devices. These model the hydro-mechanical units and servos controlling wing surfaces in the aircraft.

4.4 Modeling Synchrony and Asynchrony using AADL

We have created three versions of the TTEthernet example; a synchronous model, an asynchronous model, and a direct mode asynchronous model.

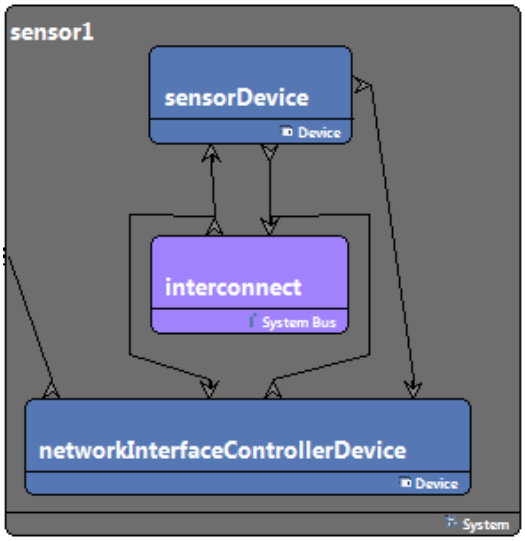

Figure 16. Sensor Component

Synchronous model: The synchronous model is a good fit for AADL modeling. Global synchrony is assumed by AADL, as well as its latency analysis plug-in. All threads are dispatched periodically, and communication delays can be expressed explicitly.

Asynchronous model: Initially, we had trouble expressing the asynchronous model, as the key assumption of global synchrony no longer holds. AADL has no concept of local clocks, or branching time as used in Computation Tree Logic (CTL) [10], thus all clock drifts must be expressed in terms of a global, linear notion of time. We have achieved this by introducing "boundary" delays between the network and components, where we add up the worst case latencies in the model.

Direct asynchronous mode: The third option we consider is a "direct mode" open-loop flight control architecture. In this configuration, the sensors and PEs are disabled, thus there is no feedback, and all control surfaces are controlled directly by the pilot. When synchronization is lost, a flight control system can quickly revert to this direct mode.

Compared to the asynchronous feedback control architecture, latencies will be lower in the direct mode control, and thus performance/latency constraints are easier to meet. The open-loop control, however, requires an airplane that is inherently stable – such as commercial transport airplanes – and is not able to control aircraft with negative or relaxed static stability – such as the F-16.

4.5 Real-time Properties: Schedulability Analysis and Latencies

In AADL, real-time properties and constraints are specified in multiple places. AADL was designed for schedulability analysis from its early stages. One can assign periods, deadlines, and dispatch policies to threads. Thus, one can quickly apply classic scheduling theory to analyze whether the AADL system is schedulable. This abstraction closely follows common assumptions of scheduling theory, such as periodic tasks, triggered independently of task dependencies, as specified by some rate.

Latency analysis in AADL builds on a different approach, added as an experimental feature in AADL v1. Latencies in AADL can be assigned to flows, connections, and buses. End-to-end flows can

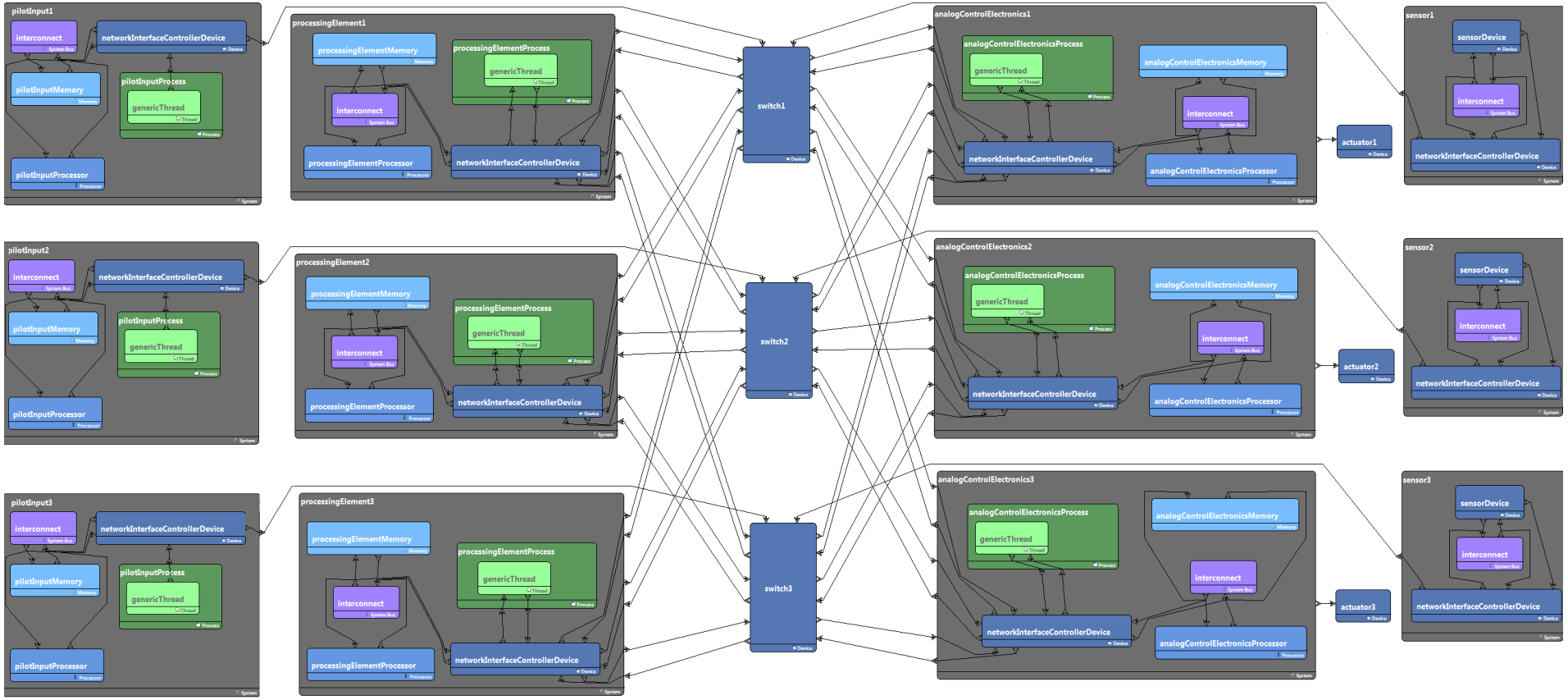

Figure 17. TTEthernet Example Modeled using AADL

then be specified by describing how multiple flows are connected to form a data flow path through the AADL model. The latency analysis can then calculate the end-to-end latency by summing the latency numbers along the path.

The schedulability analysis in AADL currently is independent of the latency analysis, and the latency analysis is independent of the schedulability analysis. Thus, changing latencies along a flow has no impact on schedulability analysis, and vice versa. Ideally, one could apply methods that automatically derive jitters for tasks as influenced by latency analysis, similar to how it is described in [11].

The modeling of asynchronous threads in AADL is another challenge. Global synchrony is a key assumption in AADL. The dispatch policy of a thread can be aperiodic, however it is not clear what triggers the thread's execution in that case. A thread may have an incoming data flow, with an associated latency. However, the semantics is not fully specified, and thus up to interpretation. One might assume event semantics for flows, saying that a thread executes whenever it receives data through an incoming flow, as specified by latencies.

Moreover, periodic threads can also be asynchronous, as they may be triggered by local clocks, that can potentially drift from each other. Since AADL assumes a globally synchronous system, however, specifying the thread dispatch policy to be periodic would imply a globally synchronous system.

As the examples above show, AADL timed semantics are ambiguous, and change depending on the context. This presents a problem when modeling mixed synchronous/asynchronous systems, as one needs to think over potentially conflicting real-time constraints. Moreover, for latency analysis jitters must be manually calculated, and specified in the models by the designers specifically. Once all such parameters are manually calculated, adding up the latency numbers is little extra value provided by AADL. By bridging the gap between the periodic task model and latency analysis, the burden of calculating jitters could be automated.

4.6 AADL State of the Art

This section describes the state of the art in AADL, as it relates to the fault and real-time analysis performed in this study. We hope that the discussion of the current AADL version will motivate a discussion, that may potentially benefit future versions of AADL.

Tasks: Threads are tasks. They support three interaction paradigms: directional flow of information through ports, access of a shared logical resource via data access, and request for services via subprogram access. In the case of directional flow of information, there are three variants: event communication, message communication with queuing, and sampling communication without queuing. These communication paradigms can be combined with the different thread dispatch semantics of time-triggered dispatch (periodic) and data-driven dispatch (aperiodic or sporadic). For example, we can represent a health monitoring system that periodically polls its alarm event queue rather than responding to each individual alarm arrival with a dispatch.

Asynchrony: According to [12], the original AADL standard defines thread dispatch and communication semantics in terms of a global clock. The Normal System Operation section of the AADL Standard[13] mentions that asynchronous system semantics are allowed. This is done by introducing additional properties and a device type to represent the clocks. AADL Version 2 explicitly introduces the notion of multiple clocks. These are referred to as synchronization domains expressed by a predeclared device type to represent the clocks. In the model, a property Reference_Time can identify the local clock.

A globally synchronous system is relevant for communication between periodic threads. In the case of aperiodic message or event communication (event data ports) the arrival of the message/event triggers the dispatch. Thus, dispatching occurs independent of clocks; it may be queued if the thread is still active with a previous dispatch.

For periodic threads there are immediate, delayed, and in AADL Version 2 sampling port connections on data ports. Immediate and delayed connections require time-based coordination of thread dispatch. For immediate connections the recipient execution start is delayed until the source thread has completed its execution. In case of delayed connections, the communication must delay the delivery of the data until the next frame, possibly by double buffering it. In other words, time-based coordination is necessary to enforce all of the semantics.

Sampling connections indicate that the recipient samples the input at its rate independent of the thread dispatch and completion. Allowing these connections to operate in both a synchronous and asynchronous system.

4.7 Results and Conclusion

The modeling work is still in progress, thus, the conclusions and inferences drawn below are a "snapshot in time". These will change as our knowledge of AADL modeling techniques improve and we get feedback from the AADL community. We have joined the AADL committee and have ongoing interactions with the principals.

So far, the modeling work has yielded a number of lessons. The difficulty in expressing protocol-centric failure behavior indicates that if the long-term goal is to use AADL models as key repositories for generating system dependability attributes, more work needs to be done. However, the promise of model-driven dependability analysis, with its potential for automated generation of system fault-trees and Failure Mode and Effects Analysis (FMEA)s, is very exciting. As the technology matures, it may allow for more systematic architectural trade-offs and safe and informed architectural optimization.

As with most model development, the ability to do system exploration of the modeled domain is a key benefit. Similarly, the ability to capture the rationale of design and the assumptions that underpin the model is of keen interest. In this regard, we found the application of the simple fault taxonomy and naming convention to be very beneficial and effective. Exploring the error modes at multiple architectural layers allowed for a more systematic *examination of conscience*, making the modeler reconsider the potential failure contributions at each layer.

An interesting side-effect of this naming convention is that such a semi-mechanical examination "checklist" yielded a potential error model state explosion as the taxonomy of error failure modes were applied to components we previously considered to be simple. For example, the modeling of a driver required decomposition into smaller sub-components to facilitate efficient modeling of concurrent failure manifestations.

Attempting to map all ingress and egress error manifestations to a single state machine rapidly became intractable, and a hierarchical decomposition of the driver was required to separate potential concurrent error contributions. For example, a single integrated-circuit quad driver yielded an error model with eight internal error state machines, as separate ingress and egress error manifestations were captured. The totality of these eight state machines were much less complex than a single 2^8 input state machine. From our experiences with the driver components, we feel that a generalized method to guide hierarchical decomposition may be beneficial (and potentially critical) if resulting models are to remain tractable for analysis.

Our work so far has been performed largely bottom-up, focusing on connectivity and protocol layers. We feel that a similar methodology would also be beneficial if applied top-down—where

application and software developers also declare the fault model for the expected communication exchanges using a similar taxonomy. Formalizing the expectations of each layer may then provide for greater application and platform re-uses; and, in the longer term, automate consistency checking of application requirements with the underlying platform and communication layer guarantees.

A key challenge we identified in modeling error propagation and mitigation is the need to compose multiple, potentially heterogeneous models of computation in order to express the behavior of both the analyzed system and error propagations. The current AADL Error Annex relies on a probabilistic automata context, whereas, AADL itself is defined using data flow-like semantics. The composition of such models must be captured for formal analysis of error propagation in BRAIN. Moreover, the behavior of the BRAIN nodes themselves must also be captured, potentially through the AADL Behavioral Annex, Finite State Machines (FSMs), or other formal languages.

Furthermore, it may become practically impossible to capture different aspects and multiple levels of abstraction in the same formal model. Reusing verification results from the formal verification of protocol functionality may help to "guide" the error propagation analysis. Probabilistic analysis lends itself especially well as a semantic domain for integrating analysis results—this may be a role for SRI's Evidential Tool Bus (ETB) integration.

To produce truly re-usable models, a better layering methodology needs to be developed. We feel that a weakness of the AADL modeling approach is that a driver model must have knowledge of the upper protocol layers. This is illustrated in the Time-triggered Protocol (TTP/C) modeling, where protocol-centric failures (i.e., those that were a function of semantic content or timing) required declarations and pass-through mappings within the driver component for protocol error propagations, even though an actual driver would have no knowledge of protocol data or time semantics. From a re-use perspective, such mappings introduce semantic layer pollution that preclude component reuse. In the ideal case, a layering hierarchy should be developed to allow for greater abstraction and pass-through of higher-layer error events. This would allow a driver model to remain agnostic to specific system $\leftrightarrow$ target instantiations.

One difficulty in developing models without an available execution and analysis environment is completeness. AADL states that an error specification is erroneous if all input propagations are not captured within a guard. Although layering the models more effectively may help here by allowing events from different layers to pass-through states without explicitly specifying them, it may also complicate the assessment of completeness.

The question of completeness is further compounded by AADL's strict ordering of guard actions. In AADL, the order of guard conditions is important, with the first matching guard taking precedence over others below it in lexical order. Although we welcome the rigors of the possible specification, we also believe that this is an area where formal model translation, simulation, and analysis (model checking) will be greatly beneficial to the modeler, ensuring that the intended behavior is what the modeler intended.

On a similar note, an issue already under discussion in the AADL working committee is the ability of a component to query the internal state of a component it is connected to. Using such coupling, it is possible to completely circumvent the AADL Error Annex error propagation mechanism and to code error transitions from coupled state knowledge.

For longer term re-use, we feel it is preferable to model events where the only permitted coupling is between components, with the possible exception of signaling an "error-free" state. The models in this report have adhered to this restriction.

In relation to the use of error probabilities, we have two findings. When discussing the more esoteric failure modes, determining the probabilities for error manifestations is sometimes closer to art than science. In an initial system model (where detailed reliability models and evidence are not available), complex failure modes are often estimated by simple rules of thumb, e.g., an expectation

of 1% of permanent failures result in babbling.

Currently, such assumptions can be modeled by adding intermediate states to the error model. However, we feel that the ability to express an event occurrence as a function of another event occurrence may be beneficial. For example, using something like:

$$occurrencefail_X = occurrencefail_Y * 0.01$$

which means that the probability of X occurring is 1% of the probability of another error event (Y). In the early states of model development, this may not require all states to sum-up to one, but we need to conduct more informal explorations to test the sensitivity to such assumptions.

A similar concept is required for hierarchical composition. By decomposing the state into separate automata, we do not want to infer that the states manifest independently. Instead, we want the probability numbers and distributions to express the failure rate of the hierarchical concurrent child automata. To express such issues in a probabilistic reasoning framework, methods must be developed that equalize probabilities as "weights," rather than treating them as hard numbers.

5 Related Work

Methods to analyze flow latency in Architecture Analysis & Design Language (AADL) models were presented in [12], the results of which were implemented in a flow latency analysis plug-in. A key assumption used throughout the study is that the execution platform is globally synchronous. Therefore, this method cannot inherently capture asynchronous communication, and reduces it to finding sampling rates for asynchronous events. This approach seems too restrictive for the analysis of asynchronous high-integrity systems. Moreover, in asynchronous systems the global Worst Case Execution Time (WCET) may not be the product of local WCET times. This, however, appears difficult to capture with the assumptions used.

An approach for system-level co-simulation of avionics systems using the Polychrony tool is shown in [14]. Polychrony is a tool based on the Signal synchronous language. Synchronous languages [15] build on the key assumptions of global synchrony, deterministic concurrency, and zero-time computation time, making them suitable for the scalable analysis of synchronous systems. As their name suggests, however, synchronous languages cannot directly express asynchronous communication.

Ocarina [16] is a tool environment developed in Ada for AADL model analysis and code generation. Ocarina uses the Cheddar tool for real-time analysis, and integrates the PolyORB middleware for code generation. The focus of Ocarina is code generation and simulations. In contrast, the methods described in this study focus on real-time and fault-tolerance analysis.

A dependability modeling framework based on AADL and Generalized Stochastic Petri Nets (GSPNs) is presented in [17]. The authors propose a translation of AADL Error Annex models to GSPN models in order to analyze reliability and availability. The work proposed in this study adopts the high-level approach of mapping AADL models to formal Model of Computations (MoCs). However, we chose Finite State Machine (FSM) as the underlying MoC for this work to overcome issues with mixed low- and high-integrity message propagation.

An approach to analyze real-time properties of AADL models using the Unified Modeling Language (UML) Modeling and Analysis of Real-Time and Embedded Systems (MARTE) profile is presented in [18]. Activity diagrams are used to capture asynchronous message passing between threads. Analytical results are then presented to obtain the end-to-end latency of the models, but the focus of the paper is to show that AADL models can be captured using UML MARTE. A method to analyze "immediate" and "delayed" data communications in AADL using UML MARTE is described in [19]. The authors define a clock constraint language that is used for manual calculations. In this report, we chose to focus on analyzing AADL models directly. Moreover, we focused on an approach that is amenable to automated analysis using Symbolic Analysis Laboratory (SAL) and the open-source Distributed Real-time Embedded Analysis Method (Dream) tool.

A method to use AADL as a specification for the performance evaluation of real-time architectures is presented in [20]. The Cheddar tool is used for the analytical and simulation-based evaluation of various communication patterns. The approach described in this paper builds on a task graph model, and is capable of achieving exhaustive state space search as described in [21]. Moreover, the token-based extensions allow the capturing of fault tolerance properties.

Schedulability analysis of AADL models using the Algebra of Communicating Shared Resources (ACSR) is presented in [22]. ACSR can be applied to perform many different scheduling algorithms, including Earliest Deadline First (EDF), Least Laxity First (LLF), etc. ACSR is also able to capture aperiodic threads, and can be analyzed using the VERSA tool [23]. The approach used in this paper is similar, as the real-time analysis is translated to reachability analysis in both cases. The SAL models developed, however, can also be used to prove fault tolerance properties.

6 Conclusion

We created an example system that, while being simple and abstract, has all the characteristics and flexibility needed to do analysis and modelling of asynchronous, synchronous, and mixed synchronous/asynchronous systems. The TTEthernet communication network used in this example system has the capability of supporting each of these timing paradigms. From this example system, we created AADL models and performed timing analysis.

In comparing two different implementations of this example system, one asynchronous and one synchronous, we found that the asynchronous model (using TTEthernet RC traffic) required an oversampling of 26 times that of the synchronous system in order to meet the 50 ms worst-case latency requirement established for this example system. This leads to a commensurate increase in both processor and bandwidth overhead. In this analysis, we found that Architecture Analysis & Design Language (AADL) did not provide the necessary queuing analysis techniques for worst-case latency calculations and these needed to be pre-specified parameters in the AADL v1 model, using an off-line queuing analysis tool.

We found that the schedulability analysis in AADL currently is independent of the latency analysis and the latency analysis is independent of the schedulability analysis. Thus, changing latencies along a flow has no impact on schedulability analysis, and vice versa. Ideally, one would apply methods that automatically derive jitters for tasks as influenced by latency analysis.

The modeling of asynchronous threads in AADL is another challenge. Global synchrony is a key assumption in AADL. The dispatch policy of a thread can be aperiodic, however it is not clear what triggers the thread's execution in that case. Periodic threads can also be asynchronous, as they may be triggered by local clocks that can potentially drift from each other. Since AADL assumes a globally synchronous system, specifying the thread dispatch policy to be periodic would imply a globally synchronous system. As AADL's semantics are ambiguous and change depending on the context, this presents a problem when modeling mixed synchronous/asynchronous systems. One needs to think over potentially conflicting real-time constraints. Moreover, for latency analysis, jitters must be manually calculated and specified in the models by the designers manually. Once all such parameters are manually calculated, adding up the latency numbers via AADL provides little extra value. By bridging the gap between the periodic task model and latency analysis, the burden of calculating jitters could be automated.

We gathered a number of other lessons about AADL from modeling communications protocols. The difficulty in expressing protocol-centric failure behavior indicates that if the long-term goal is to use AADL models as key repositories for system dependability attributes, more work needs to be done to mature AADL's capability and ease-of-use in this area. As the technology matures, analysis tools linked to AADL may allow for more systematic architectural trade-offs and safe and informed architectural optimization.

We found the application of the simple fault taxonomy and naming convention we developed to be very beneficial and effective in trying to model failures in and/or propagated by communication mechanisms. Exploring the error modes at multiple architectural layers allowed for a more systematic *examination of conscience*, making the modeler reconsider the potential failure contributions at each layer.

However, attempting to map all ingress and egress error manifestations of a communication channel to a single state machine per the AADL Error Annex rapidly became intractable and a hierarchical decomposition of the communication's media driver was required to separate potential concurrent error contributions. From our experiences with the driver components, we feel that a generalized method to guide hierarchical decomposition may be beneficial (and potentially critical) if resulting models are to remain tractable for analysis. Furthermore, it may become

practically impossible to capture different aspects and multiple levels of abstraction in the same formal model. Reusing verification results from the formal verification of protocol functionality may help to "guide" the error propagation analysis. Probabilistic analysis lends itself especially well as a semantic domain for integrating analysis results—this may be a role for SRI's Evidential Tool Bus (ETB) integration.

When examining rare and complex failure modes, determining the probabilities for error manifestations is sometimes closer to art than science. In an initial system model (where detailed reliability models and evidence are not available), complex failure modes are often estimated by simple rules of thumb, e.g., an expectation of 1% of permanent failures result in babbling. We feel that the ability to express an event occurrence as a function of another event occurrence may be beneficial. For example, using something like:

$$occurrencefail_X = occurrencefail_Y * 0.01$$

which means that the probability of X occurring is 1% of the probability of another error event (Y).

A similar concept is required for hierarchical composition. By decomposing the state into separate automata, we do not want to infer that the states manifest independently. Instead, we want the probability numbers and distributions to express the failure rate of the hierarchical concurrent child automata. To express such issues in a probabilistic reasoning framework, methods must be developed that equalize probabilities as "weights," rather than treating them as hard numbers.

References

1. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis and Design Language (AADL) Annex Volume 1. SAE Standards N^o AS5506A, June 2006.

2. *ITU-T: ITU-T Recommendation G.701: General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms*. Telecommunication Standardization Sector of ITU, 1993.

3. ARINC: *ARINC Specification 659: Backplane Data Bus*. Aeronautical Radio, Inc, dec 1993.

4. Flexray Consortium: FlexRay Communications System Protocol Specification Version 2.1 Revision A. *Control*, , no. 2.1, 2005.

5. Int. Standardization Organisation, ISO 11898-4: *Road vehicles - Controller area network (CAN) – Part 4: Time-triggered communication*. 2004.

6. SAE, A.-D. E. C. S. C.: Time-Triggered EThernet. SAE Standards N^o AS6802A, June 2011.

7. Hall, B.; Paulitsch, M.; and Driscoll, K.: FlexRay BRAIN Fusion: A FlexRay-Based Braided Ring Availability Integrity Network. *SAE World Congress*, , no. Paper No 2007-01-1492, 2007.

8. Hall, B.; Paulitsch, M.; Benson, D.; and Behbahani, A.: Jet Engine Control Using Ethernet with A BRAIN. *44th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, no. AIAA 2008-5291, 21 - 23 July 2008, Hartford, CT, 2008.

9. Paulitsch, M.; and Hall, B.: Insights into the Sensitivity of the BRAIN (Braided Ring Availability Integrity Network)–On Platform Robustness in Extended Operation. *Dependable Systems and Networks, International Conference on*, vol. 0, 2007, pp. 154–163.

10. Clarke, E. M.; and Emerson, E. A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs, Workshop*, 1982, pp. 52–71.

11. Tindell, K.; and Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, vol. 40, April 1994, pp. 117–134.

12. Peter Feiler, J. H.: Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). , CMU/SEI Technical Note CMU/SEI-2007-TN-010, 2007.

13. AS-2 Embedded Computing Systems Committee SAE: Architecture Analysis & Design Language (AADL). SAE Standards N^o AS5506A, January 2009.

14. Yu, H.; Ma, Y.; Glouche, Y.; Talpin, J.-P.; Besnard, L.; Gautier, T.; Guernic, P. L.; Toom, A.; and Laurent, O.: System-level Co-simulation of Integrated Avionics Using Polychrony. *Proceedings of the 2011 ACM Symposium on Applied Computing*, ACM, New York, NY, USA, 2011, pp. 354–359.

15. Benveniste, A.; Caspi, P.; Edwards, S. A.; Halbwachs, N.; Guernic, P. L.; Robert; and Simone, D.: The Synchronous Languages 12 Years Later. *Proceedings of The IEEE*, 2003, pp. 64–83.

16. Lasnier, G.; Zalila, B.; Pautet, L.; and Hugues, J.: Ocarina: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications. *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 237–250.

17. Rugina, A.-E.; Kanoun, K.; and Kaâniche, M.: Architecting Dependable Systems IV. Springer-Verlag, Berlin, Heidelberg, A System Dependability Modeling Framework using AADL and GSPNs, 2007, pp. 14–38.

18. Mallet, F.; and de Simone, R.: *MARTE vs. AADL for Discrete-Event and Discrete-Time Domains* , Springer, vol. 36 of *LNEE*, 2. April 2009, pp. 27–41.

19. André, C.; Mallet, F.; and de Simone, R.: Modeling of immediate vs. delayed data communications: from AADL to UML MARTE. *Proceedings of ECSI Forum on specification & Design Languages (FDL)*, 2007, pp. 249–254.

20. Dissaux, P.; and Singhoff, F.: Stood and Cheddar: AADL as a Pivot Language for Analysing Performances of Real Time Architectures. *Proceedings of 4th International Congress on Embedded Real-Time Systems*, 2008.

21. Madl, G.; Dutt, N.; and Abdelwahed, S.: Performance Estimation of Distributed Real-time Embedded Systems by Discrete Event Simulations. *Proceedings of EMSOFT*, 2007.

22. Sokolsky, O.; Lee, I.; and Clarke, D.: Schedulability analysis of AADL models. *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IEEE Computer Society, Washington, DC, USA, 2006, pp. 179–179.

23. Clarke, D.; Lee, I.; and liang Xie, H.: VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. *Journal of Computer and Software Engineering*, vol. 3, 1995.

Appendix A

Acronyms and Initialisms

AADL Architecture Analysis & Design Language

ACE Analog Control Electronics

ACSR Algebra of Communicating Shared Resources

BRAIN Braided Ring Availability Integrity Network

CTL Computation Tree Logic

DES Discrete Event Simulation

DREAM Distributed Real-time Embedded Analysis Method

EDICT Error Detection Isolation Containment Types

EDF Earliest Deadline First

ETB Evidential Tool Bus

FMEA Failure Mode and Effects Analysis

FSM Finite State Machine

GSPN Generalized Stochastic Petri Net

IMA Integrated Modular Avionics

LLF Least Laxity First

MAC Medium Access Control

MARTE Modeling and Analysis of Real-Time and Embedded Systems

MoC Model of Computation

NIC Network Interface Controller

OSATE Open Source AADL Tool Environment

PE Processing Element

RC rate-constrained

SAE Society of Automotive Engineers

SAL Symbolic Analysis Laboratory

TT time-triggered

TTP/C Time-triggered Protocol

UML Unified Modeling Language

WCET Worst Case Execution Time

Appendix B

DREAM Model of TTEthernet Case Study

The AFCS-Distributed Systems project has been established on NASAs DASH*link* to support public dissemination of the models and results of this program. The URL https://c3.nasa.gov/dashlink/ will take the user to the Home Page of DASH*link*. The user can access the site by hovering over the RESEARCH AREAS pull-down list and right clicking on Verification and Validation. Scroll down to AFCS Distributed Systems and right click the project. Right click Distributed Real-time Embedded Analysis Method (DREAM) under Popular Resources. The DREAM model is contained in the file ttethernet_ dream_ model.txt

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 09 - 2012	Contractor Report	October 2010 - March 2011

4. TITLE AND SUBTITLE

Modeling and Analysis of Mixed Synchronous/Asynchronous Systems

5a. CONTRACT NUMBER

NNL10AB32T

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Driscoll, Kevin R.; Madl, Gabor; Hall, Brendan

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 23681-2199

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-2012-217765

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

Practical safety-critical distributed systems must integrate safety critical and non-critical datain a common platform. Safety critical systems almost always consist of isochronous components that have synchronous or asynchronous interface with other components. Many of these systems also support a mix of synchronous and asynchronous interfaces. This report presents a study on the modeling and analysis of asynchronous, synchronous, and mixed synchronous/asynchronous systems. We build on the SAE Architecture Analysis and Design Language (AADL) to capture architectures for analysis. We present preliminary work targeted to capture mixed low- and high-criticality data, as well as real-time properties in a common Model of Computation (MoC). An abstract, but representative, test specimen system was created as the system to be modeled.

15. SUBJECT TERMS

AADL; Asynchronous interface; Communication network; Synchronous interface

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	43	(443) 757-5802