Application of Software Health Management Techniques*

Nagabhushan Mahadevan
Institute for Software Integrated Systems
Vanderbilt University
Nashville TN

Abhishek Dubey
Institute for Software Integrated Systems
Vanderbilt University
Nashville TN

Gabor Karsai
Institute for Software Integrated Systems
Vanderbilt University
Nashville TN

ABSTRACT

The growing complexity of software used in large-scale, safety critical cyber-physical systems makes it increasingly difficult to expose and hence correct all potential defects. There is a need to augment the existing fault tolerance methodologies with new approaches that address latent software defects exposed at runtime. This paper describes an approach that borrows and adapts traditional 'System Health Management' techniques to improve software dependability through simple formal specification of runtime monitoring, diagnosis, and mitigation strategies. The two-level approach to health management at the component and system level is demonstrated on a simulated case study of an Air Data Inertial Reference Unit (ADIRU). An ADIRU was categorized as the primary failure source for the in-flight upset caused in the Malaysian Air flight 124 over Perth, Australia in 2005.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Fault tolerance]; D.2.0 [**Software Engineering**]: General—*Protection mechanisms*; D.2.4 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems, Real-time systems and embedded systems*

General Terms

Reliability, Design, Management, Experimentation

Keywords

Fault Diagnosis and Mitigation, Real-Time Systems

*This paper is based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. The authors thank Dr Paul Miner, Eric Cooper, and Suzette Person of NASA LaRC for their help and guidance on the project.

1. INTRODUCTION

Due to the increasing software complexity in modern cyber-physical systems there is a likelihood of latent software defects that can escape the existing rigorous testing and verification techniques but manifest only under exceptional circumstances. These circumstances may include faults in the hardware system, including both the computing and non-computing hardware. Often, systems are not prepared for such faults. Such problems have led to a number of failure incidents in the past, including but not limited to those referred to in these reports: [26, 6, 7, 18].

State of the art for safety critical systems is to employ software fault tolerance techniques that rely on redundancy and voting [23, 34, 9]. However, it is clear that existing techniques do not provide adequate coverage for problems such as common-mode faults and latent design bugs triggered by other faults. Additional techniques are required to make the systems self-managing, i.e. they have to provide resilience to faults by adaptively mitigating faults and failures.

Self-adaptive systems, while in operation, must be able to adapt to latent faults in their implementation, in the computing and non-computing hardware; even if they appear simultaneously. Software Health Management (SHM) is a systematic extension of classical software fault tolerance techniques that aims at implementing the vision of self-adaptive software using techniques borrowed from System Health Management for complex engineering systems. System health management typically includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline) [27, 20]. Note that SHM can be considered as a dynamic fault removal technique [5]. It is performed at run-time, and it includes detection, isolation, and mitigation actions to remove fault effects. System health management also includes prognostics and possibly Software Health Management can be extended in that direction as well, but we have not investigated it yet.

Our research group has been involved in developing tools and techniques, including a hard real-time component framework built over the platform services provided by ARINC-653 compliant operating systems [1], for software health management [15, 16]. The core principle behind our approach is the thesis that it is indeed possible to deduce the behavioral dependencies and failure propagation across an assembly of software components, if the interactions between those components are restricted and well-defined. Here, components imply software units that encapsulate parts of a software system and implement specific service(s). Simi-

lar approaches can be found in [12, 35]. The key difference between those and our work is that we apply an online diagnosis engine coupled with a two-level mitigation scheme.

In this paper, we provide a description and discussion of our work with respect to a case study that approximately emulates the working of an Air Data Inertial Reference Unit (ADIRU) used on Boeing 777 aircraft. Our goal is to show how an SHM architecture can be used to detect, diagnose, and mitigate the effects of component level failures such that the system-wide functionality is preserved. This work extends our previous works [15, 16] to allow multi-module systems working on different physical computers. We also extended the detection functionality developed earlier for monitoring the correctness of data on all ports to enable observers that can also monitor the sequence of activities inside a component. We have created the necessary software infrastructure to close the loop from detecting an anomaly and diagnosing a component failure to issuing the necessary mitigation actions in real-time.

Paper Outline: Section 2 reviews related research, section 3 describes the motivating example: the incident caused by a software malfunction and the ADIRU architecture. Section 4 presents the concepts and capabilities of the software component framework. Sections 5-6 describe the implemented case study and explain our approach to software health management. We conclude with a discussion of results and a summary. Note that the paper focuses on the case study. Details of the technological background are available in other papers [15, 14, 16, 13].

2. RELATED RESEARCH

The work described here fits in the general area of self-adaptive software systems, for which a research roadmap has been presented in [10]. Our approach focuses on latent faults in software systems, follows a component-based architecture, with a model-based development process, and implements all steps in the Collect/Analyze/Decide/Act loop.

Rohr et al. advocate the use of architectural models for self-management [30]. They suggest the use of a runtime model to reflect the system state and provide reconfiguration functionality. From a development model they generate a causal graph over various possible states of its architectural entities. At the core of their approach, they use specifications based on UML to define constraints, monitoring and reconfiguration operations at development time.

Garlan et al. [17] and Dashofy et al. [11] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections, and properties of interest. Their work follows the theme of Rohr et al., where architectural models are used at runtime to track system state and make reconfiguration decisions using rule-based strategies.

While these works have tended to the structural part of the self-managing computing components, some have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [40]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. They model the source and target models for the program using state charts and then specify an adaptation model, i.e., the model for the adaptation set connecting the source model to the target model using a variant of Linear Temporal Logic [39].

Williams' research [29] concentrates on model-based autonomy. The paper suggests that emphasis should be on developing techniques to enable the software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[37] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [38] for guiding the system to the desirable behaviors.

Lately, the focus has started to shift to formalize the software engineering concepts for self-management. In [22], Lightstone suggested that systems should be made "just sufficiently" self-managing and should not have any unnecessary complicated function. Shaw proposes a practical process control approach for autonomic systems in [31]. The author maintains that several dependability models commonly used in autonomic computing are impractical as they require precise specifications that are hard to obtain. It is suggested that practical systems should use development models that include the variability and unpredictability of the environment. Additionally, the development methods should not pursue absolute correctness (regarding adaption) but should rather focus on the fitness for the intended task, or sufficient correctness. Several authors have also considered the application of traditional requirements engineering to the development of autonomic computing systems [8, 33].

The work described here is closely related to the larger field of software fault tolerance: principles, methods, techniques, and tools that ensure that a system can survive software defects that manifest themselves at run-time [24], [28]. Arguably, our approach comes closest to dynamic software fault removal, performed at run-time. The overall architecture presented below shows a specific implementation of the functions needed to perform this task.

3. CASE STUDY: THE ADIRU

In 2005, the flight computer of Malaysian Air flight 124 - a Boeing 777, flying to Kuala Lumpur from Perth registered excessive acceleration values in all three body axes - vertical acceleration changed to -2.3g within 0.5 seconds, lateral acceleration decreased to -1.01g within 0.5 second and the longitudinal acceleration increased to +1.2g within 0.5 second. As a result, the flight computer pitched the aircraft up and commanded it to a steep climb. Thereafter, the airspeed decreased and the aircraft descended. Re-engagement of autopilot was followed by another climb of 2,000 ft. The investigation report [6] revealed that the problem was caused due to an anomaly in the fault masking software in the aircraft's primary *Air Data Inertial Reference Unit* (ADIRU). An ADIRU provides airspeed, angle of attack, altitude as well as inertial position and attitude information to other flight systems. To understand the scenario we need to briefly summarize the ADIRU architecture.

ADIRU Architecture: The primary design principle in Boeing 777's ADIRU Architecture [25, 32] is multiple levels of redundancy. There are two ADIRU units: primary and secondary. The primary ADIRU is divided into 4 Fault Containment Areas (FCA), with each FCA containing multiple Fault Containment Modules (FCM): accelerometers (6 FCM), gyros (6 FCM), processors (4 FCM), power supplies (3 FCM), ARINC 629 bus (3 FCM). The ADIRU system

was designed to be serviceable, with no need of maintenance with one fault in each FCA. Systems can still fly with two faults, but it necessitates maintenance upon landing. A secondary unit, the S(econdary)AARU also provided inertial data. The flight computer compares the data received from primary unit and secondary unit before using it.

Accelerometers and gyros are arranged on the face of a dodecahedron in a skewed redundant configuration [25]. Thus, any four accelerometers and gyros are sufficient to calculate the linear acceleration in the body inertial reference frame and angular velocity in the fixed frame of reference. This calculation is replicated in parallel by each one of 4 processors.

Failure Analysis: Post-flight analysis [6] revealed that in 2001 accelerometer 5 had failed with high output values and was subsequently marked as faulty. However, because there was only one failure no maintenance was requested on the unit, but the status of failed unit was recorded in on-board maintenance memory. However, on the day of the incident, a power cycle on the primary ADIRU occurred, during flight. Upon reset, the processors did not check the status of the on-board memory and hence did not regard accelerometer 5 as faulty. Thereafter, a second in-flight fault was recorded in the accelerometer 6 and was disregarded. Till the time of the incident the ADIRU processors used a set of equations for acceleration estimation that disregarded the values measured by accelerometer 5. However, the fault in accelerometer 6 necessitated a reconfiguration to use a different set of estimation equations. At this point, they allowed the use of accelerometers 1 to 5 as accelerometer 5 was not regarded as faulty, passing the abnormal high acceleration values to all flight computers. Due to common-mode nature of the fault, voters allowed the incorrect accelerometer data to go out on all channels. This high value was used by primary flight computers, although a comparison function used by the flight computers lessened the effect. In summary, a latent software bug and the common-mode nature of the accelerometer fault bypassed the redundancy checks and caused the effect to cascade into a system failure [19].

In the rest of this paper, we will show that such problems can be avoided by augmenting the redundancy-based fault protection by a real-time health management framework that can perform system-level detection, diagnosis, and mitigation. To demonstrate our approach we emulated the necessary components[1] of the ADIRU using the hard real-time ARINC-653 Component Framework [15].

4. THE ARINC COMPONENT MODEL

Our approach to Software Health Management is based on a component-oriented architecture that assumes a component model that includes strict rules for component interactions. For our research we have defined a specific component model: the ARINC-653 Component Model (ACM) that is implemented by a runtime software layer called the ARINC Component Framework (ACF). ACM borrows concepts from other software component frameworks, notably from the CORBA Component Model (CCM) [36], and is built upon the capabilities of ARINC-653 [1]: the state of the art operating system standard used in Integrated Modular Avionics. A key concept in ARINC-653 is spatial and temporal isolation among partitions, where partitions host computational processes.

In ACM, a component can have four kinds of external ports for interactions: **publishers, consumers, facets** (provided interfaces[2]) and **receptacles** (required interfaces). Each port has an interface type (a named collection of methods) or an event type (a structure). The component can interact with other components through **synchronous** call/return interfaces (assigned to provided or required ports), and/or via **asynchronous** publish/subscribe event connections (assigned to publisher and consumer ports). Additionally, a component can host internal methods that are periodically triggered.

Unlike CCM frameworks, where the functional logic belonging to a port is executed on a new or pre-existing but dynamically allocated worker-thread, here the port's functional logic is statically bound to a unique ARINC-653 process. Therefore, each port can be periodic (i.e. time triggered), or aperiodic (i.e. event triggered). This binding is defined and configured during initialization. Given that a provided interface can have more than one method, every method is allocated to a separate process. At any time, only one process per component is allowed to be in the running state, thus each process must obtain a component lock before it can execute. During design, the developers must identify the real-time properties for each component port, including frequency, deadline, worst case execution time etc.

Model-based design: ACF comes with a modeling language built using our model integrated computing tools (`http://www.isis.vanderbilt.edu/research/MIC`) that allows the developers to model a component and the set of interfaces it provides independent of the actual deployment configuration. This enables a preliminary, constraint-based analysis of the system. Such an analysis can be used to check, for instance, type compatibility among connected ports. The model captures the component's real-time properties and resource requirements using a domain specific modeling language. System integrators then configure software **assemblies** specifying the architecture of the system built from interacting components.

The **deployment configuration** consists of processors uniquely mapped to ARINC-653 modules, with each module containing one or more partitions. These partitions are temporally and spatially isolated. System integrators map one or more components in the assembly to a partition.

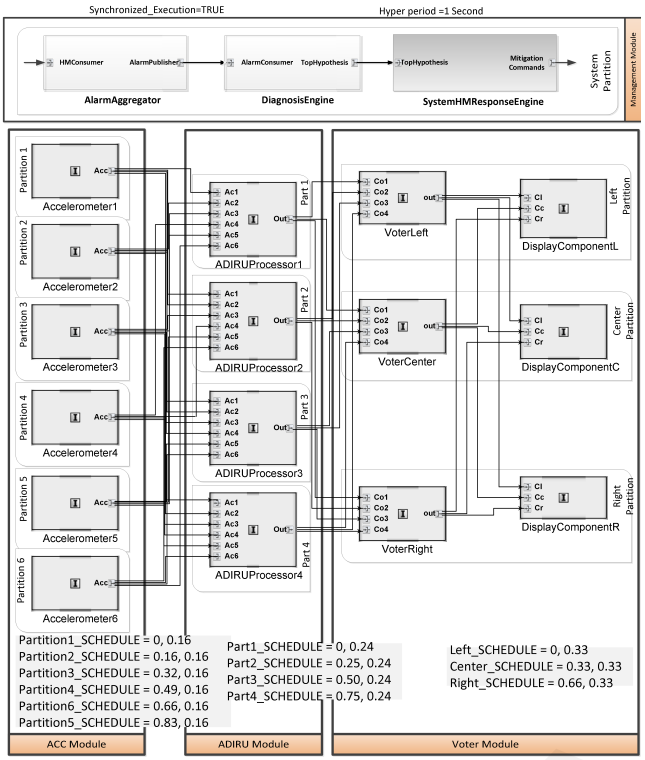
Component developers can also specify local monitors and local health management actions for each component (described later using the case study example). Once the assembly has been specified, system integrators are required to specify the models for system-level health management (described later).

Software Health Management in ACF: Within the framework there are various levels at which health management techniques can be applied, ranging from the level of individual components or the level of subsystems to the whole system. In the current work, we have focused on two levels of software health management: *component level* that is limited to the component, and the *system level* that includes global information for performing diagnosis to identify the root failure mode(s) and components.

Component-Level Health Management: (CLHM) pro-

[1]We did not emulate the gyros and actual flight control logic.

[2]An interface is a collection of related methods.



Component	Port	Period	Deadline
Accelerometer	Acc	1sec	1sec
Processor	Ac1-Ac6	-1sec	1sec
Processor	Out	-1sec	1sec
Voter	Co1-Co4	-1sec	1sec
Voter	Out	-1sec	1sec
Display	Cl,Cc,Cr	-1sec	1sec

Figure 1: Model for the ADIRU Assembly. -1 denotes an aperiodic process.

vides localized and limited functionality for managing the health of one component by detecting anomalies, mitigating its effects using a reactive timed state machine – on the level of individual components. It also reports to the higher-level, system health manager.

System-Level Health Management: (SLHM) manages the overall health of the system i.e. assembly of all components. The CLHM processes hosted inside each of the components report their input (monitored alarms) and output (mitigation actions) to the SLHM. It is important to know the local mitigation action because it could affect how the faults cascade through the system. Thereafter, the SLHM is responsible for the identification of root failure source(s), with multiple failure mode hypotheses being handled. Once the fault source is identified, appropriate mitigation strategy is employed.

Code generation: Code generation tools allow the integrators to generate the glue code (to realize component interactions), and code for the health management. Relieving the software developer from the arduous task of writing code for implementing interactions ensures that we can restrict the semantics, so that we can analyze the system failure propagation at design time, before deployment. The generated code includes the wrappers necessary to launch and configure the ARINC-653 ports associated with the component. These wrappers follow a strict template for each kind of port: checking pre conditions, acquiring locks (to ensure atomic operation on the component), invoke developer written functional code, and checking post conditions. Developers write the functional code using only the exposed interfaces provided by the framework. When an anomaly is detected by a monitor, it is always reported to the local component health manager. Deadline violation is always monitored in parallel by the underlying framework.

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} -0.3717 & -0.3386 & 0.8644 \\ 0.3717 & -0.8644 & 0.3386 \\ -0.6015 & -0.7971 & -0.0536 \\ -0.9732 & 0.1625 & 0.1625 \\ -0.6015 & -0.0536 & -0.7971 \\ 0.2298 & -0.6882 & -0.6882 \end{bmatrix} \times \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} + \mathcal{N}$$

Table 1: Accelerometer Equations

5. MODELING FOR ACM

This section describes how we modeled the ADIRU software architecture using the ACM Modeling Language, in order to conduct experiments. We did not model the gyros in this example, and timing used does not reflect the actual timing on the real system.

Software Assembly: Fig. 1 shows the different components that are part of this example. This figure also shows the management module, which implements the system-wide health manager. Also shown are the real-time properties for the ports of each type of component. We will cover the components in that module in detail in subsequent sections. Different parts of the assembly are organized into modules. Each ARINC-653 module is deployed on a different host processor. The modeling paradigm also captures the internal data flow and control flow of the components, not shown in the figure. This is required to create the fault propagation graph as discussed later in section 6.4.

There are six instances of accelerometer components. Each accelerometer component has a periodic publisher that publishes its data every 1 second. The published data consists of a linear acceleration value measured in the axis of the accelerometer and a time stamp. All accelerometers measure in directions perpendicular to the six faces of a dodecahedron centered at the origin of the body coordinate system. Table 1 shows the equation relating measured acceleration, a_1, to a_6 in terms of three orthogonal body acceleration vectors, a_x,a_y,a_z. Here $\mathcal{N}$ is a 6×1 vector of zero mean, Gaussian noise. Running the model interpreter of the ACM framework generates the code for all accelerometers. The only portion supplied by the developer is the function that is called in every cycle to produce the data. We use a lookup table to simulate actual sensor measurements, configured for each experiment.

All acceleration values are fed to the four ADIRU processors, which process the values measured by the six accelerometers and solve a set of linear regression equations to estimate the body linear acceleration. Each ADIRU processor consists of six aperiodic consumers and a periodic publisher. It should be noted that if a processor is aware of a fault in one of the accelerometers it can ignore that particular observation and use the other 5 for performing regression. The following equations present the predicted acceleration values along the body axes, derived by solving the regression equations using all six accelerometers : $\hat{a_x} = -0.19a_1 + 0.19a_2 - 0.30a_3 - 0.49a_4 - 0.30a_5 + 0.11a_6$, $\hat{a_y} = -0.17a_1 - 0.43a_2 - 0.40a_3 + 0.08a_4 - 0.03a_5 - 0.34a_6$, and $\hat{a_z} = +0.43a_1 + 0.17a_2 - 0.03a_3 + 0.08a_4 - 0.40a_5 - 0.34a_6$. The output of each ADIRU processor is the body axis data

and is published every second to the three voter components. The voters consume these data with three consumers. Each voter uses a median algorithm to choose the middle values and outputs it to the display component.

Deployment: Fig. 1 also shows the deployment. Each accelerometer is deployed on a separate partition in an ARINC-653 module. Module schedule is also shown. ADIRU processors are deployed on 4 partitions on one module. A pair of a voter and a display unit shares a single partition on the last module. The ACF ensures that all modules run in a synchronized manner with the specified system-wide hyper period of 1 second. At the start of each hyper period a controller sends a synchronization message to each module manager, which executes the module schedule. This is similar to the technique in the TTP/A protocol [21].

6. SOFTWARE HEALTH MANAGEMENT

As briefly discussed in section 4, we use a two-level approach for implementing a software health management framework: (a) component level with local view of the problem, and (b) the system level. The component level health management deals with detecting violations and taking local mitigation action within a component. The system level health management deals with aggregating the data (monitor and local mitigation action) from component health managers across the entire system, performing a system-wide diagnosis to identify the fault-source and taking a system-wide mitigation action based on the results of the diagnosis. The following sub-sections discuss these aspects with respect to the ADIRU example more detail.

6.1 Component Level Anomaly Detection

The ACM framework allows the system designer to deploy monitors which can be configured to detect deviations from expected behavior, violations in properties, constraints, and contracts of an interaction port or component. Table 2 describes the different discrepancies that can be observed on a component port and the component as a whole. A detailed description is provided in the paper [16]. While the monitors associated with resource usage are run in parallel by framework, other monitors are evaluated in the same thread executing the component port. When any monitor reports a violation, the status is communicated to its **Component Level Health Manager (CLHM)** and then possibly to the **System Level Health Manager (SLHM)**.

In addition to the monitors described in Table 2, new monitors have been introduced that inform the component health manager of the current component-process (port) being executed. These monitors report an *ENTRY* into and *EXIT* from a process. These are used to observe the execution sequence using an observer state machine and thereby detect and/or prevent any deviations that might adversely affect the health/operation of the component.

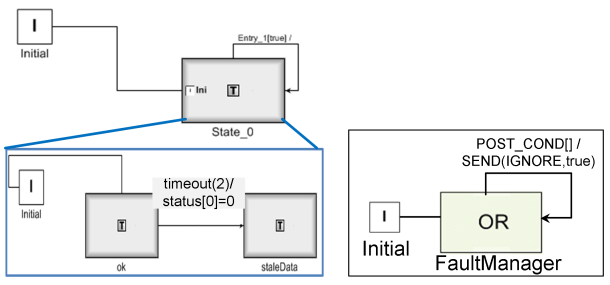
Monitors in the Modeled ADIRU Assembly: In the ADIRU assembly, the monitors are configured to track the resource usage (CPU time) of the publishers / consumers in the Components associated with Accelerometers, ADIRU processors, Voters and Display components. The publisher port in each Accelerometer component is configured with a monitor to observe the published data via a post condition. These monitors fire if the published data from the associated Accelerometer appears to be Stuck-High or Stuck-Low or show a rapid change in value that is more than the established norms. All the consumer ports in each of the ADIRU-processors, Voters and Display components have a specified Data-Validity time and the associated monitors trigger when the age of the incoming data (i.e. the difference between the current time and the timestamp on the data) is more than the specified Data-Validity time. Another set of monitors are configured to check for violations of a pre-condition for the consumer ports in the Display component. This property detects rapid changes in the data fed to these consumers consistent with the physical limits on aircraft acceleration and jerk (rate of change of acceleration) in each one of the body axes.

<**PreCondition**>::=<Condition>
<**PostCondition**>::=<Condition>
<**Deadline**>::=<double value> */* from the start of the process associated with the port to the end of that method */*
<**Data_Validity**>::=<double value> */* Max age from time of publication of data to the time when data is consumed*/*
<**Lock Time Out**>::=<double value> */* from start of obtaining lock*/*
<Condition>::=<Primitive Clause><op><Primitive Clause>|<Condition><logical op><Condition>| !<Condition> | True| False
<Primitive Clause>::=<double value>| Delta(Var)| Rate(Var)|Var */* A Var can be either the component State Variable, or the data received by the publisher, or the argument of the method defined in the facet or the receptacle*/*
<op>::= < | > | <= | >= | == | !=
<logical op>::=&& | ||

Table 2: Monitoring Specification. Comments are shown in italics.

HM Action	Semantics
CLHM: IGNORE	Continue as if nothing has happened
CLHM:ABORT	Discontinue current operation, but operation can run again
CLHM: USE_PAST _DATA	Use most recent data (only for operations that expect fresh data)
CLHM: STOP	Discontinue current operation Aperiodic processes (ports): operation can run again Periodic processes (ports): operation must be enabled by a future START HM action
CLHM: START	Re-enable a STOP-ped periodic operation
CLHM RESTART	A Macro for STOP followed by a START for the current operation
SLHM: RESET	Stop all operations, initialize state of component, clear all queues. start all periodic operations
SLHM: STOP	Stop all operations

Table 3: CLHM and SLHM Mitigation Actions.



(a) (b)

Figure 2: (a) Accelerometer 1 Observer inside the ADIRU processor. (b) CLHM State-Machine of Accelerometer1

In addition to the monitors specified above, the ADIRU

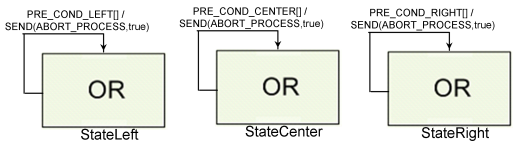


Figure 3: CLHM associated with Display Component. PRE_COND events are generated when a pre-condition is violated. SEND is an action that sends the mitigation to failed process.

processor components look for the absence of published data on each of the consumer ports, connected to one of the six accelerometers. This is done by observing the lack of the *ENTRY/EXIT* events from these ports within a pre-specified timeout period, see Fig. 2(a). It shows portions of the state machine specification monitoring the events for accelerometer 1. Once a missing data is detected, the status is set to 0. The `status` array, indexed from 0 and having six elements, captures the state of all six channels. Five other, similar state machines are used for observing the other accelerometers, in parallel.

6.2 Component Level Mitigation

Once a discrepancy is detected, the generated code provided by the framework reports the problem to the CLHM. The ACM modeling language allows the CLHM to be defined as a timed-state machine that acts upon input error/discrepancy/anomaly events and outputs the appropriate local mitigation action. The CLHM for each component is deployed on a high-priority ARINC process that is triggered when any of the associated processes (that host the Component ports) or the underlying ACM framework report any violations detected by monitors. In a blocking call, the reporting process waits for a response/mitigation action from the CLHM. Table 3 lists the mitigation actions that can be issued by the CLHM.

Additionally, the CLHM can be configured to take on the additional responsibility of an *observer*. As an observer, the CLHM state machine uses the input events detected by *ENTER* and *EXIT* monitors to track the execution sequence for the component ports, possibly together with the evolution of the component's state. Such tracking can detect incorrect sequencing of component operations, or illegal states of the component. When any deviation is observed, the observer can trigger the health manager portion of the CLHM state machine to take the appropriate mitigation action, and/or transition to a new state.

CLHM in the ADIRU assembly: Components associated with an Accelerometer and a Display host a CLHM. In case of the Accelerometers, the CLHM, see Fig. 2(b), is configured to issue an IGNORE command when the post-condition violation is detected in the publisher. In case of the Display component, the CLHM, see Fig. 3, has a parallel state machine to observe and manage faults detected in the consumers associated with left, right, and center channels. Each of these parallel machines responds with an ABORT command if a pre-condition violation is observed in the data input to the consumer. As described in the previous section, this pre-condition checks whether the rate of change of acceleration does violate the specifications. In both cases, the CLHM reports the anomaly detected and the local mitigation command issued to the System Level Health Manager.

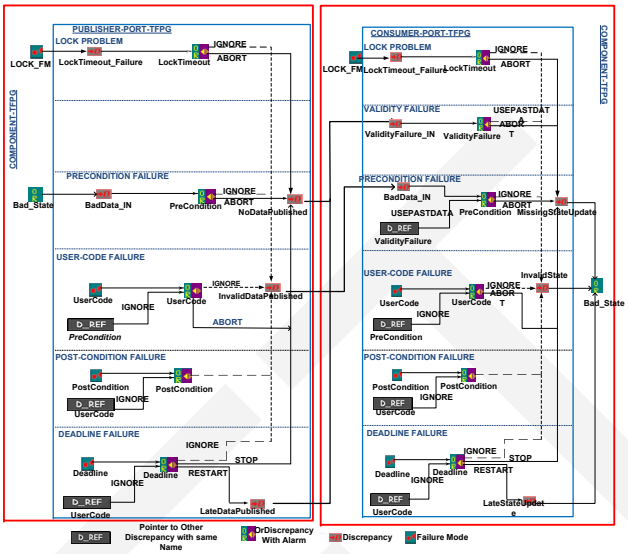


Figure 4: TFPG: Publisher/Consumer interaction

6.3 System-Level Health Management

While component level health management is performed by the CLHM inside the Component, the system level health management requires additional, system-wide components. These new components: **Alarm Aggregator**, **Diagnosis Engine**, and **SystemHMResponse Engine** have dedicated tasks associated with System Health Management. Fig. 1 shows these additional System Level Health Management components, hosted in a separate module, for the ADIRU assembly.

The Alarm Aggregator is responsible for collecting and aggregating inputs from the component level health managers (local alarms and the corresponding mitigation actions). It hosts an aperiodic consumer that is triggered by the data (alarm, and local mitigation command) provided by the Component Level Health Managers. The Alarm Aggregator assimilates the information received from the CLHM-s in a moving window (two hyper periods long) and sorts them based on their time of occurrence. This data is fed to the Diagnosis Engine. This engine uses a model-based reasoner to diagnose the source of the fault by searching for an explanation for the alarms collected by the Alarm Aggregator. Finally, the SystemHMResponse Engine component acts upon the diagnosis result to deliver the appropriate system level mitigation response.

In order to interact with the System Level Health Management components, each functional component in the existing Assembly model is provided an additional publisher: **HMPublisher**, and consumer: **HMConsumer**. The publisher is used by the Component Health Manager to feed local detection and mitigation data to the Alarm Aggregator. The consumer is used to receive commands from the SystemHMResponse Engine. To avoid clutter, Fig. 1 does not show these additional publishers and consumers.

6.4 System Level Diagnosis

To identify the fault-source, the Diagnosis Engine component in the SLHM needs to reason over the alarms (and their associated local mitigation actions) received from one or more Component Health Managers. The reasoning process isolates the fault source using a diagnosis technique based on

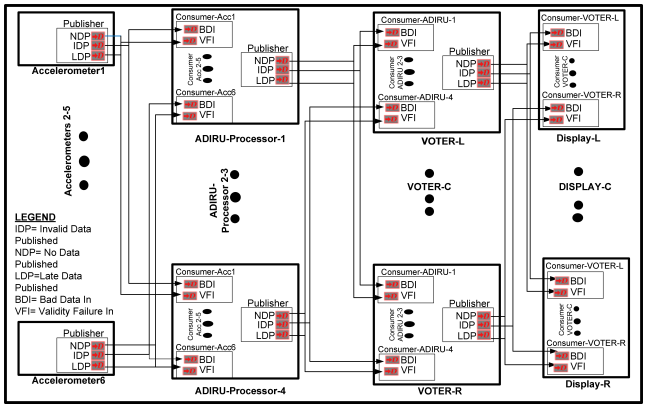


Figure 5: TFPG model of the ADIRU system

a **Timed-Failure Propagation Graph (TFPG)** model of the system. In a TFPG model [2, 3, 4] the fault-sources (Failure Modes) and the anomalies (observed or un-observed Discrepancies) of the system are represented as nodes of a labeled, directed graph. The directed edges between nodes capture the propagation of the failure effect from the source node (Failure Mode/Discrepancy) to the destination node (Discrepancy). A propagation timing interval and the system mode wherein the fault effect can propagate are captured as edge properties.

Automatic Synthesis Of Fault Propagation Graph: In this work, the TFPG model of the system is auto generated using the information available in the system's ACM model. The TFPG model of the system is made up of the TFPG model of its associated component, which in turn is made up of the TFPG model of the interaction ports (Publisher / Consumer/ Provides/ Requires ports) present in that component. As each of the ports follow a well-defined sequence of operations, a specific TFPG-template model can be created for each of these types. The template TFPG model contains the Failure-Modes, Discrepancies and the failure-propagation edges specific to that ACM-port type. The TFPG model of each component is populated with instances of the appropriate template-TFPG model (based on the type of ACM-ports contained in the component). The data-flow and the control flow graph model of a Component, is useful in identifying additional failure propagation edges within the Component. The interaction captured in the Assembly model (e.g. Fig. 1) helps in identifying failure propagation interactions across two components.

Diagnosis: The generated TFPG-model is used by the diagnosis engine to hypothesize the fault-source(s) that could have triggered a specific set of alarms. While additional alarms certainly help in narrowing the fault-source, it is possible that the observed alarm set (observable discrepancies) could be explained by multiple hypotheses (fault source). Thresholds based on hypothesis metrics such as **Plausibility** and **Robustness** [4] are used to prune the hypotheses set. Further, the component containing the most number of fault-sources (as identified by the pruned hypotheses set) is categorized as the faulty component.

TFPG Model of ADIRU Assembly: The TFPG model for the ADIRU system was auto-generated using the approach described above. This section explains the generation process and the TFPG models in more detail. Fig. 4 captures an instance of the template TFPG model of a publisher and a consumer. Additionally it captures the Failure Propagation effect between the publisher and consumer.

As previously stated, each of the component port types has a set of generic operations performed in a well-defined sequence in each cycle of execution. During this process, a set of monitors are invoked to detect anomalies. Currently these monitors detect violations and problems related to Lock-Acquire, Data-Validity (in consumers), Pre-Condition and Post-Condition checks, errors in User-Code and Deadline violations. The TFPG model of the Publisher and Consumer port in the Fig. 4 shows the Discrepancies associated with these monitors and the failure propagation interaction of these Discrepancies with other Discrepancies and Failure modes. These failure propagations correspond to the cascading effects of failures within the ACM-port as well as the failure propagation into or out of the component port (here Publisher/Consumer).

Discussion of Fault Propagation: As shown in the TFPG for the publisher-port, it is evident that inability to acquire the Component Lock prevents the Publisher code from running, thereby resulting in no data being published (see section 4 for description of the generated code for all ports.). Another fault-propagation example includes a 'Bad Input' to the Publisher port that could lead to pre-condition violation which in-turn could lead to different kinds of anomalies based on the CLHM's local mitigation action. An ABORT command issued by CLHM for a pre-condition violation could again lead to the problems associated with no data being published. An IGNORE CLHM command could resolve the issue (with no further alarms) or possibly cascade into a user code anomaly and/or a post-condition anomaly and/or a deadline violation. The net result of these failures could be either no data being published or invalid data being published or the data being published late. All these effects could affect the consumer downstream. Again, a part of the scenario described above could be triggered even with good data input into the publisher. It is possible that in this case there is no pre-condition violation, but a fault in the user code (captured by USER_Code failure mode) could trigger a set of anomalies leading to down-stream problems associated with the published data.

It can be seen that the TFPG model of the consumer is mostly similar to that of the publisher. This is because the generic operations triggered in sequence during an execution/run of the consumer are similar to that of the publisher. The difference is there because the consumer consumes a data-token and updates certain state variables. This results in the failure effects propagating out of the consumer port affecting the state variables that the consumer updates (No_Update, Invalid_Update, and Late_Update).

Intra-Component Failure Propagation: Since the ACM ports are hosted inside specific components, the failure effects could propagate (in either direction) between the ports (i.e. within the component) and between components. With reference to the Fig. 4, it can be seen that the publisher is affected if the component state-variable (used by the publisher) is affected. This is represented as a failure-propagation between the Bad_State discrepancy in the component TFPG and the Bad_Data_IN discrepancy in the publisher TFPG. The failure effects of a bad-data input affecting the publisher's pre-condition, or user-code evaluation is captured by the failure propagation links in the publisher TFPG. In the case of the consumer-port, the failure effects originate from the consumer and affect the state variables

(updated by the consumer) in the component.

Inter-Component Failure Propagation: Interaction between the publisher and the consumer is captured in the Assembly model. This implies that the output discrepancies in the publisher can possibly propagate failure effects to the input discrepancies on the consumer side. For example, the failure effects associated with either no data published or data published late could affect the validity of the data consumed in the consumer. Alternatively, an invalid data published from the publisher could lead to a pre-condition or user code violation in the consumer. As can be seen from the TFPG model described in Fig. 4, a Bad_State introduced in the Publisher component, could cascade through the publisher to the consumer side and the associated states of the consumer component.

System TFPG: Fig. 5 captures the component-level TFPG model for the entire ADIRU assembly model. The details of the TFPG model of the publisher/consumer ports and their interaction with their respective components is not shown in this model and is considered to be hidden within the component and ACM port TFPG models. This generated TFPG model is used by the reasoner in the Diagnosis Engine component. When new data is received from the Alarm Aggregator component, the reasoner generates a set of hypotheses that best describe the cause for the alarms. As new alarms are received it updates the hypotheses. The hypotheses with the best metric (Plausibility, Robustness) are regarded as the most plausible explanation. Further, if a system-level mitigation strategy is specified, then the component containing the source failure modes is identified and the information is passed on to the component hosting the system-level mitigation strategy: the SystemHMResponse Engine.

6.5 System Level Mitigation

The system-level mitigation strategy is modeled using hierarchical timed state machine formalism in the ACM modeling language. This state machine is executed inside the SystemHMResponse Engine component. This component has an aperiodic consumer that receives the diagnosis results from the Diagnosis-Engine Component. Upon arrival of a new diagnosis result, the consumer triggers the state machine implementing the system level mitigation with the input event: the diagnosis result. If a mitigation action needs to be taken on a specific component, the state machine's output is sent to the appropriate component. The commands issued by the System-Level Health Manager are RESET, STOP, CHECKPOINT and RESTORE commands to faulty components (see table 3). Currently, the System Level Health Manager action is considered for the entire component (i.e. all ARINC processes in the Component).

Mitigation Strategy for ADIRU: The system-level mitigation strategy for the ADIRU is currently modeled as a hierarchical parallel timed state machine. This is the first method we have chosen to solve the problem, but other techniques, possibly involving reasoning and search could also be applicable. A benefit of using a timed FSM-s is that they allow (some level of) verification via model checking. Fig 6 captures the mitigation strategy for each Accelerometer fault. An initial command is issued to RESET the Accelerometer component, hoping that this will get the Accelerometer to work correctly. If despite the reset, the same Accelerometer is identified as a fault-source within a specified time-limit, then command is issued to STOP the Accelerometer.

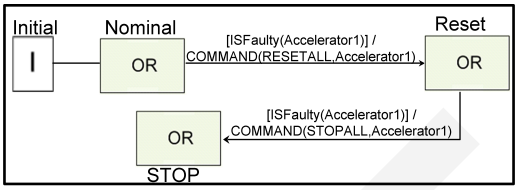


Figure 6: Portion of the Mitigation Strategy (State Machine) dealing with Accelerometer1

7. EXPERIMENTS AND DISCUSSION

We deployed the three different modules of the ADIRU assembly shown in Fig. 1 on three computers in our lab. These computers were running our ARINC-653 emulator on top of Linux, and had the ARINC Component runtime. The computers were deployed in an isolated subnet, with the network shared by all hosts. Upon initialization, all modules synchronized with the system module that ran the diagnoser and system response engine. Thereafter, each module cyclically scheduled its partitions. All modules resynchronized with the system module at the start of each hyper period. The code necessary for this distributed synchronization was auto generated from the ADIRU deployment model in which each module was mapped to a physical core on a processor.

Table 4 shows the highlights of the events as they were recorded throughout the system. Time is relative to the first event. All faults, including accelerometer 6 and 5 were artificially injected and turned on after a fixed number of iterations. From our observations we noticed that our diagnoser was correctly able to determine that the problem was caused by accelerometer 5 and shut it down. Thereafter, the redundancy management algorithm in the ADIRU processor was able to reconfigure itself to use a different set of regression equations that did not use Accelerometer 5 or 6, and prevented a system-wide failure.

8. CONCLUSION

Self-adaptive systems, while in operation, must be able to adapt to latent faults in their implementation, in the computing and non-computing hardware; even if they appear simultaneously. Software Health Management (SHM) is a systematic extension of classical software fault tolerance techniques that aims at implementing the vision of self-adaptive software using techniques borrowed from system health management. SHM is predicated on the assumptions that (1) specifications for nominal behavior are available for a system, (2) a monitoring system can be automatically constructed from these specifications that detect anomalies, (3) a systematic design method and a generic architecture can be used to engineer systems that implement SHM.

In this paper we have presented our initial work towards such an SHM approach. We heavily rely on our model-based technologies (domain-specific modeling languages, software generators, and model-based fault diagnostics), but we believe the overhead caused by this apparatus is worthwhile, as the designer can directly work with specifications and design the (SHM) system on a high level. Our experiments have illustrated the approach but its large-scale, industrial application still remains.

The SHM technique described above is only the first step towards the vision and much work remains. For example,

Table 4: Event Sequence

Time(s)	Module:Component	Description
-	Accelerometer-5	Accelerometer 5 is known to be faulty and not being used by the processors. Accelerometer 5 post-condition violation followed by IGNORE from its CLHM. No pre-condition violation in the display components. SLHM diagnosis does not produce a hypothesis that crosses robustness threshold.
Power Reset of ADIRU-Processors		
0	ADIRU Processor(s)	Reset. During initialization fail to read information that Accelerometer-5 is faulty. But continue to use only Accelerolmeter1,2,3,4,6
Accelerometer-6 fails silent		
1.00	Accelerometer-6	Fails Silently. Detected by the observers in ADIRU Processors.
2.00	ADIRU-Processor(s)	Reconfigure the set of regression equations - end up using Accelerometer-5.
Using Faulty Accelerometer-5		
2.74	Accelerometer-5	Published Bad Data. Post-condition check violated.
Local Mitigation & Report to SLHM		
2.74	Accelerometer-5 CLHM	Receives Post-Condition violation alarm and issues an IGNORE command. Passes the data to Alarm Aggregator.
3.25	Alarm Aggregator	Receives data from Accelerometer-5. Buffers and later sends it to Diagnosis Engine.
Faulty Data Consumed & Processed		
4.00	4 ADIRU-Processors (1-4)	All of them use data from faulty Accelerometer-5. Hence results from all ADIRU-Processors are skewed
4.25	3 Voters (Left, Center, and Right)	Voters compute the results based on the ADIRU-Component outputs. They cannot isolate the faulty data as all the input-data (to Voter) display is similarly affected.
Local Mitigation & Report to SLHM		
4.25	3 Display Components (Left, Center, Right)	Consume data fed by the voters. Data show pre-condition violation.
4.25	Display Component(s) CLHM	Receives Pre-Condition violation alarm and issues an ABORT command. Passes the data to Alarm Aggregator.
4.25	Alarm Aggregator	Receives data from Display Component. Buffers and later sends it to Diagnosis Engine.
System Level Health Management - Alarm Aggregation, Diagnosis, Mitigation		
5.25	Alarm Aggregator	Feeds data to Diagnosis Engine.
5.25	Diagnosis Engine	Receives alarm data from Alarm Aggregator and runs the TFPG-Reasoner. Detects Accelerometer-5 to be a possible fault candidate. Supporting alarms received from Pre-Condition violations in Display-Components increases the metric and confirms the fault in Accelerometer-5. Feeds result to Response Engine Comp component to take mitigation action
5.39	Response Engine Comp	Receives information on the faulty component - Accelerometer-5 - and issues command to reset.
5.74	Accelerometer-5	Receives command to Reset from Response Engine Comp. Resets itself.
Reset of Accelerometer-5 has no effect		
6.74	Accelerometer-5	Fault in Accelerometer-5. Published Bad Data. Violates Post-Condition check.
Local Mitigation & Report to SLHM		
6.84	Accelerometer-5 CLHM	Receives Post-Condition violation alarm and issues an IGNORE command. Passes the data to Alarm Aggregator.
6.95	Alarm Aggregator	Receives data from Accelerometer-5. Buffers and later sends it to Diagnosis Engine.
Faulty Data Consumed & Processed & Other monitors trigger		
7	4 ADIRU Processor (1-4)	All ADIRUs use all the Accelerometers (including the faulty Accelerometer-5). Hence results from all ADIRU-Processors are skewed
7.25	3 Voters (Left, Center, and Right)	Voters compute the results based on the ADIRU-Processor outputs. They cannot isolate the faulty data as all the input-data (to Voter) display is similarly affected.
7.30	3 Display Components (Left, Center, Right)	Consume data fed by the voters. Pre-condition violations detected
Local Mitigation & Report to SLHM		
7.30	Display-Component(s) CLHM	Receives Pre-Condition violation alarm and issues an ABORT command. Passes the data to Alarm Aggregator.
7.34	Alarm Aggregator	Receives data from Display-Component(s). Buffers and later sends it to Diagnosis Engine.
System Level Health Management - Alarm Aggregation, Diagnosis, Mitigation		
8.30	Alarm Aggregator	Feeds data to Diagnosis Engine.
8.65	Diagnosis Engine	Receives alarm data from Alarm Aggregator and runs the TFPG-Reasoner. Again detects Accelerometer-5 to be a possible fault candidate. Finds other monitors/alarms that support the hypothesis. Feeds result to Response Engine Comp component to take mitigation action
8.65	Response Engine Comp	Receives information on the faulty component - Accelerometer-5 - and issues command to STOP it.
8.95	Accelerometer-5	Receives command to Stop from Response Engine Comp. Stops itself.
Post Stopping Accelerometer 5		
9.45	ADIRU-Processor(s)	Over-time ADIRU-Processors detects using the observer that there is no data from Accelerometer-5. Stop using Accelerometer-5. Regression equations use accelerometers 1-4
9.75	Display-Component(s)	The Data received from the Voter(s) do not violate the Pre-condition. Back to Healthy operation.

fault management systems need to be verified to show that they do not violate safety rules. The verification of such adaptive systems is a major challenge for the research community. Furthermore, the approach described is based on reactive state machines that encode the strategies to handle specific failure modes. Designers have to model these reactions explicitly. In a more advanced system, a deliberative, reasoning-based approach can be envisioned that derives the correct reaction based on some high-level goals and the current state of the system. Such an advanced approach to SHM is currently being investigated.

9. REFERENCES

[1] ARINC specification 653-2: Avionics application software standard interface part 1 - required services.

[2] S. Abdelwahed and G. Karsai. Notions of diagnosability for timed failure propagation graphs. In *Proc. IEEE Systems Readiness Technology Conference*, pages 643–648, Sept. 2006.

[3] S. Abdelwahed, G. Karsai, and G. Biswas. A consistency-based robust diagnosis approach for temporal causal systems. In *16th International Workshop on Principles of Diagnosis*, pages 73–79, 2005.

[4] S. Abdelwahed, G. Karsai, N. Mahadevan, and S. C. Ofsthun. Practical considerations in systems diagnosis using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, February 2009.

[5] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004.

[6] A. T. S. Bureau. In-flight upset; 240km NW Perth, WA; Boeing Co 777-200, 9M-MRG. August 2005.

[7] A. T. S. Bureau. AO-2008-070: In-flight upset, 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. October 2008.

[8] D. W. Bustard and R. Sterritt. A requirements engineering perspective on autonomic systems development. *Autonomic Computing: Concepts, Infrastructure, and Applications*, pages 19–33, 2006.

[9] R. Butler. A primer on architectural level fault tolerance. Technical report, NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108, 2008.

[10] E. Cheng, Betty H. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[11] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[12] R. de Lemos. Analysing failure behaviours in component interaction. *Journal of Systems and Software*, 71(1-2):97 – 115, 2004.

[13] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. Towards a real-time component framework for software health management. ISIS-09-111, Institute for Software Integrated Systems, Vanderbilt University, Nov 2009. `www.isis.vanderbilt.edu/sites/default/files/TechReport2009.pdf`.

[14] A. Dubey, G. Karsai, and N. Mahadevan. Towards model-based software health management for real-time systems. ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University, August 2010. `http://isis.vanderbilt.edu/sites/default/files/Report.pdf`.

[15] A. Dubey, G. Karsai, and N. Mahadevan. A Component Model for Hard-Real Time Systems: CCM with ARINC-653. *Softw., Pract. Exper.*, 2011. To Appear. Draft available at `http://isis.vanderbilt.edu/sites/default/files/Journal_0.pdf`.

[16] A. Dubey, G. Karsai, and N. Mahadevan. Model-based software health management for real-time systems. In *Aerospace conference, 2011 IEEE*, march 2011.

[17] D. Garlan, S. W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, 2003.

[18] W. S. Greenwell, J. Knight, and J. C. Knight. What should aviation safety incidents teach us? In *SAFECOMP*, 2003

[19] C. Johnson, C.W.;Holloway. The dangers of failure masking in fault-tolerant software: Aspects of a recent in-flight upset event. In *2nd IET Systems Safety Conference*, pages 60–65. 2007.

[20] S. Johnson, editor. *System Health Management: With Aerospace Applications.* John Wiley & Sons, Inc, To Appear in 2011.

[21] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. In *ISORC*, pages 16 –23, 2000.

[22] S. Lightstone. Seven software engineering principles for autonomic computing development. *ISSE*, 3(1):71–74, 2007.

[23] M. R. Lyu. *Software Fault Tolerance*, volume New York, NY, USA. John Wiley & Sons, Inc, 1995.

[24] M. R. Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering*, pages 153–170, 2007.

[25] M. D. W. Mcintyre and D. L. Sebring. Integrated fault-tolerant air data inertial reference system, 1994.

[26] NASA. Report on the loss of the mars polar lander and deep space 2 missions. Technical report, NASA, 2000.

[27] S. Ofsthun. Integrated vehicle health management for aerospace platforms. *Instrumentation Measurement Magazine, IEEE*, 5(3):21 – 24, Sept. 2002.

[28] L. L. Pullum. *Software fault tolerance techniques and implementation.* Artech House, Inc., USA, 2001.

[29] P. Robertson and B. Williams. Automatic recovery from software failure. *Commun. ACM*, 49(3):41–47, 2006.

[30] M. Rohr, M. Boskovic, S. Giesecke, and W. Hasselbring. Model-driven development of self-managing software systems. In *"Models@run.time" at (MoDELS/UML)*, 2006.

[31] M. Shaw. Self-healing: Softening precision to avoid brittleness. In *Proceedings of the first workshop on Self-healing systems*, pages 111–114,2002.

[32] M. Sheffels. A fault-tolerant air data/inertial reference unit. In *Digital Avionics Systems Conference, 1992. Proceedings., IEEE/AIAA 11th*, pages 127 –131, Oct. 1992.

[33] A. Taleb-Bendiab, D. W. Bustard, R. Sterritt, A. G. Laws, and F. Keenan. Model-based self-managing systems engineering. In *DEXA Workshops*, pages 155–159, 2005.

[34] W. Torres-Pomales. Software fault tolerance: A tutorial. Technical report, NASA, 2000.

[35] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electron. Notes Theor. Comput. Sci.*, 141(3):53–71, 2005.

[36] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the CORBA component model. *Component-based software engineering: putting the pieces together*, pages 557–571, 2001.

[37] B. Williams, B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[38] B. C. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.

[39] J. Zhang and B. H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005.

[40] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006.