Fault-Adaptivity in Hard Real-Time Component-Based Software Systems*

Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37203, USA

Abstract. Complexity in embedded software systems has reached the point where we need run-time mechanisms that provide fault management services. Testing and verification may not cover all possible scenarios that a system encounters, hence a simpler, yet formally specified run-time monitoring, diagnosis, and fault mitigation architecture is needed to increase the software system's dependability. The approach described in this paper borrows concepts and principles from the field of 'Systems Health Management' for complex aerospace systems and implements a novel two level health management architecture that can be applied in the context of a model-based software development process.
At the first level, the Component-level Health Manager (CLHM) provides localized and limited service for managing the health of individual software components. A higher-level System-level Health Manager (SLHM) manages the health of the overall system. SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model automatically synthesized from the system specification built in the model-based design environment that accompanies the runtime system. SLHM also includes a reactive timed state machine used for mitigation, whose code is also generated from the model-based specification. This paper uses simple examples to illustrate the use of the approach.

1 Introduction and Motivation

Software has become the key enabler for a number of core capabilities and services in modern systems [28]. For example, a modern car contains around 20 million lines of code, while just the flight control software of modern aircraft like F-22 and F-35 contains $1.7 - 5.7$ million lines of code [9]. Given the scale of the software systems, it is not hard to appreciate the challenge of ensuring correct behavior, especially in avionics where software malfunctions have caused a

* 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 would like to thank Dr Paul Miner, Eric Cooper, and Suzette Person of NASA LaRC for their help and guidance on the project.

number of incidents in the past, including but not limited to those referred to in these reports: [5,6,18,29]. [36] provides an excellent discussion on the complexity in avionics software.

The state of the art for critical software development includes process standards such as DO-178B [12] and the emerging standards such as DO-178C [21]. However, it is known that software can contain latent defects or bugs that can escape the existing rigorous testing and verification techniques and 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.

State of the art for safety critical systems is to employ software fault tolerance techniques that rely on redundancy and voting [8,25,40]. 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 the functional effects of those faults.

Self-adaptive systems must be able to adapt to faults in software as well as the hardware (physical equipment) elements of a system, even if they appear simultaneously. Conventional Systems Health Management is associated with the physical elements of the system, and includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (at runtime/ online during operation), maintenance (offline), and fault prognostics (online or offline) [22,30]. Software Health Management (SHM), borrows concepts and techniques from Systems Health Management and is a systematic extension of classical software fault tolerance techniques. Srivastava and Schumann provide a good motivation for Software Health Management in [38]. SHM is performed at run-time, and just like Systems Health Management it includes detection, isolation, and mitigation to remove fault effects. SHM can be considered as a dynamic fault removal technique [4]. While Systems Health Management also includes prognostics, Software Health Management could possibly be extended in that direction as well, but we have not investigated it yet.

We have developed an approach and model-based support tools for implementing software health management functions for component-based systems. The foundation of the architecture is a real-time component framework that defines a component model for ARINC-653 systems[1] [14]. This framework brings the concept of temporal isolation, spatial isolation, strict deadlines from ARINC-653 and merges it with the well-defined interaction patterns described in CORBA Component Model [42]. The health management in the framework is performed at two levels. The Component-level Health Manager (CLHM) provides localized and limited service for managing the health of individual software components. A

[1] ARINC-653 (Avionics Application Standard Software Interface) is a specification for space and time partitioning in Safety-critical avionics Real-time operating systems. It allows to host multiple applications of different software levels on the same hardware in the context of an Integrated Modular Avionics architecture.[1,32]

higher-level System Health Manager (SLHM) manages the health of the overall system.

SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) model automatically synthesized from the component assembly; the engine reasons about fault effect cascades in the system, and isolates the fault source components. This is possible because the data / behavioral dependencies and hence the fault propagation across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework. Once the fault source is isolated, the necessary system level mitigation action is taken. Similar approaches can be found in [23,41]. The key difference between those and our work is that we apply an online diagnosis engine coupled with a two-level mitigation scheme. Furthermore, this approach is applied to hard real-time systems where all processes run within finite time bounds and are continuously monitored for deadline violations. This includes, the health management processes.

Our approach is supported by a model-based design environment where developers can create models of the system and its components, as well as specify how fault mitigation will take place. A suite of software generators produce glue code that allows developer-supplied functional code or 'business logic' to form a collection of applications that run on an ARINC-653 platform. The novel contributions of our approach are:

- Model-based development of component-based systems for ARINC-653 platform.
- Automatic synthesis of monitoring code that is executed with the component operations.
- Automatic synthesis of diagnosis information from the system design models.
- Automatic synthesis of the mitigation code based on system specification.
- Generation and configuration of the distributed architecture required to operate the components in parallel with the component and system level health managers.

This paper is an extended version of the work presented in [15,27]. It uses simple examples to describe the approach. A larger case study of applying the SHM principles to an Inertial Measurement Unit is available as a technical report [16]. In this paper, the focus is on the mitigation aspects: the support provided in the framework and modeling language. The outline of this paper is as follows: Sections 2 discusses the related research. Overview of the component model and design tools is given in Section 3. Section 4 presents component health manager and system-level health manager. Finally we conclude with discussions and future work.

2 Related Research and Background

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. In this context of health management, this would imply *Collect* details about anomalies observed), *Analyze* the data collected to identify/ diagnose the fault candidate / *Decide* on the possible mitigation command and finally *Act* to implement the mitigation commands.

Rohr et al. advocate the use of architectural models for self-management [35]. 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 [46]. 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 [45].

Williams' research [34] 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)[43] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [44] 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 [24], 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 [37]. 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 consid-

ered the application of traditional requirements engineering to the development of autonomic computing systems [7,39].

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 [26,33]. 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 Overview of ARINC-653 Component Model

Systems health management and fault tolerance approaches are based on the notion of interacting components. Hence, it is natural to apply this concept to SHM, where the software is built from components that can be individually developed, monitored and managed at run-time. In our work, the first step was to develop and implement such a component model. The ARINC-653 component model (ACM) is built upon the services of ARINC-653; an avionics standard for safety critical operating systems [1]. ARINC-653 systems group *processes*[2] into spatially and temporally separated *partitions*, with one or more partitions assigned to each *module* (i.e. a processor), and one or more modules forming a *system*.

Spatial partitioning ensures exclusive use of a memory region by an ARINC-653 partition. It also guarantees that a faulty process in a partition cannot ruin the data structures of other processes in other partitions, isolating low-criticality vehicle management components from safety-critical flight control software components. Temporal partitioning ensures exclusive use of the processing resources by a partition. A fixed periodic schedule is used by the RTOS to share the resources between partitions. This deterministic scheduling ensures that each partition is allowed exclusive access to the processor or other hardware resources within its predetermined execution interval. It also guarantees that when the predetermined execution interval of a partition is over, the partition's execution will be interrupted, the partition will be placed into a dormant state and the next partition in the schedule order will be granted exclusive access to the computing resource, i.e. the processor.

The ARINC-653 component model allows the developers to group a number of ARINC-653 processes into a reusable component. A component is a group of processes that share state but they do not interact directly. However, components do interact with each other via well-defined interaction patterns (chosen from a fixed set), facilitated by ports. In ACM, a component can have four kinds of external ports for interactions: **publishers, consumers, facets** (provided interfaces[3]) and **receptacles** (required interfaces), see Figure 1. Each port has an interface type (a named collection of methods) or an event type (a data

[2] An ARINC-653 process is a unit of concurrency that is analogous to thread in a desktop operating system such as Linux.

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

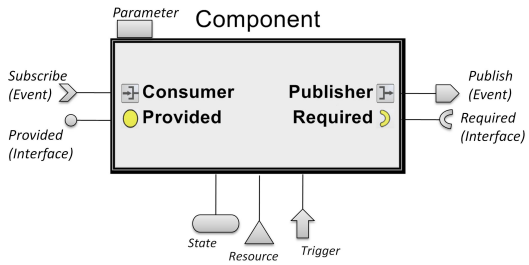


Fig. 1. The Component Model

structure). The component can interact with other components through **synchronous** call/return interfaces (associated with facets or receptacles), 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. Most of these interactions borrow concepts from other software component frameworks, notably from the CORBA Component Model (CCM) [42]. The component model also provides guidance on the allocation of activities to a component.

Real-Time Properties ACM components differ from classical CCM component in a number of ways. The underlying operating system layer on which ACM is built is geared towards hard real-time systems. Therefore, in ACM all processes have fixed properties that are specified and fixed at the system configuration time - these properties include, period, deadline, stack size and priority. Furthermore, a process can have two kinds of deadlines, HARD deadline and SOFT deadline. A HARD deadline violation is an error that is handled at the system level by the health management framework, discussed later in the paper. A soft deadline violation results in warnings. Due to these restrictions, it is not possible to dynamically assign component operations or ports to an ARINC-653 system at runtime. Therefore, all ports are statically bound to an ARINC-653 process and no dynamic memory allocation is allowed. Furthermore, the access to component state is synchronized by a component-wide lock. Priority inversion issues do not arise because all processes of a component are executed at the same priority. Please see [14] for detailed description.

The framework implementing ARINC-653 component model consists of two parts (a) a Linux-based runtime environment, and (b) a modeling environment and associated design tools. Together these tools allow systems to be developed in two distinct phases. The first phase is completed by the component developer. A component is a reusable artifact that provides one or more functionalities. It can be developed and hosted in a repository for reuse. Often, the component developer can organize various components into subsystems. The second phase is completed by the system integrator. The system integration includes the modeling and configuring of the system architecture, deploying the components on computing hosts, etc. This phase is assisted by a suite of model-driven tools.

3.1 Component Development

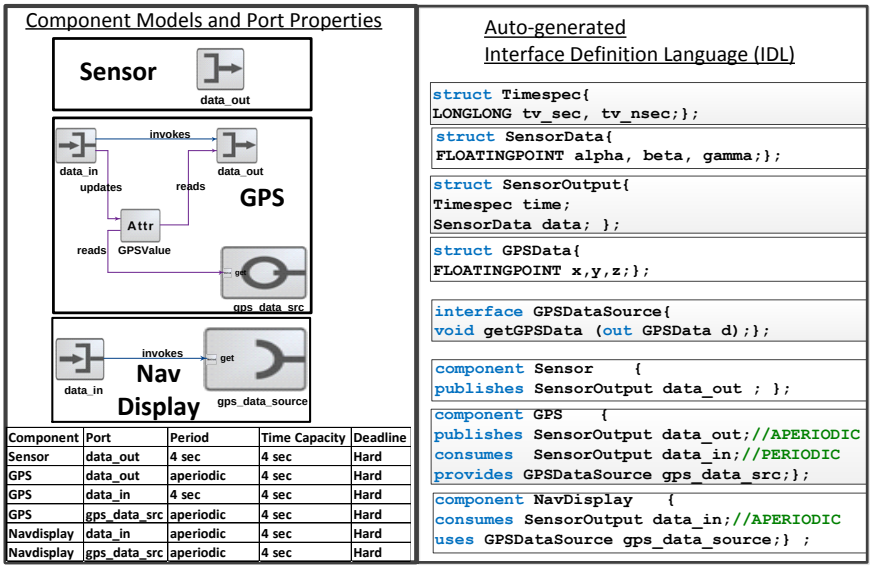
The model-based design tools[4] allow the developer to design the components. The first step in designing a component involves defining its interfaces, i.e. the ports associated with the component. Each port, as described earlier, should belong to one of the four categories: publisher, consumer, provided, required. The publisher and consumer ports need to be associated with an event (data) type, while provided and required ports need to be associated with an interface type. Furthermore, each port needs to be configured with properties related to its real-time execution: periodicity, deadline, worst case execution time, etc.

The development environment provides tool-support to bring the bare-bones component model to life. As a first step, a C++ class is generated corresponding for each component, with methods corresponding to each port. The developer is provided with specific regions in the generated code to insert the necessary code and logic to customize the behavior of each port (as per the associated task). The generated code acts as the 'glue' between the underlying ACM framework and the user-specified code to support the operation/execution of each of the component ports as dedicated ARINC-653 processes.

The component model can be further enriched by specifying the conditions that must be satisfied for each execution of the port (and its associated ARINC-653 process). These conditions are divided into three categories: pre-conditions, invariants, and post-conditions. The design tools generate monitoring code that is used to ensure and validate the correctness of these conditions during runtime. Any violation of these conditions is considered an **anomaly**. More discussion on this topic will be provided later in Section 3.2. Each component developer can also specify a local mitigation activity: a component level health manager that takes local corrective actions when an anomaly is detected. Once fully specified, the component model captures the component's interaction ports, conditions associated with the ports, the real-time properties and resource requirements of the ports and the component, the data and control flow within the component, and (optionally) the local component level health management strategy.

Example Figure 2 shows three components developed in ACM modeling environment. It also shows portion of the Interface Definition Language (IDL) file generated by the associated tools. The first component is the sensor component that publishes a data type "SensorOutput" periodically every 4 sec. The second component is the GPS component that receives the input from a Sensor, then filters it, and updates its internal data structure. It publishes the updated information through a port aperiodically. The GPS has the ability to be queried remotely via a method call for the current GPS value. The last component is a Navigation Display component, which receives an updated SensorOutput and also queries a remote GPS interface. It should be noted that the components described were developed in isolation, i.e. they are developed as part of a distributed system.

[4] These tools are available for download from `https://wiki.isis.vanderbilt.edu/mbshm/index.php/Main_Page`



Component	Port	Period	Time Capacity	Deadline
Sensor	data_out	4 sec	4 sec	Hard
GPS	data_out	aperiodic	4 sec	Hard
GPS	data_in	4 sec	4 sec	Hard
GPS	gps_data_src	aperiodic	4 sec	Hard
Navdisplay	data_in	aperiodic	4 sec	Hard
Navdisplay	gps_data_src	aperiodic	4 sec	Hard

Fig. 2. Components developed using ACM Design Tools. This figure contains the internal ports of the components, including the internal data flow and control flow. Also shown are the snapshots of generated Interface Definition Language (IDL) files and the associated real-time properties for each port.

3.2 Component Execution and Failure Scenarios

Any component, once deployed in the system can be in one of the following three states: **active**, **inactive** and **semi-active**. When a component is in inactive state, none of the ports in the Component perform their task. The active state of a component is the exact opposite of the inactive, and all the component ports perform their task. In a semi-active state, only the Consumer and Receptacle ports of a component are operational, the Publisher and Provided ports are disabled. During nominal operation, a component is either in the active state, or semi-active state. The semi-active state is typically assigned to passive replicas, if any, in the system by the system integrator. Typically, a component is made inactive only if it is diagnosed as faulty at runtime.

While the component is executing i.e. it is in active or semi-active state, component ports can introduce faults in the system. We consider two root failure sources for each component port (a) a concurrency fault: caused by the timeout in the act of obtaining the lock associated with the component, (b) a latent defect in the code written by the developer for handling the activity of the port.

Both of the above fault scenarios can lead to several secondary anomalies in either the same component or in a connected component. In our framework, the design tools allow the system designer to specify monitors which can be configured to detect deviations from expected behavior, violations of specifications

and conditions of an interaction port or component. Based on these monitors, following discrepancies can be currently identified:

- *Lock timeout*: The framework implicitly generates monitors to check for resource starvation. Each component has a lock (to avoid interference among callers), and if a caller does not get through the lock within a specified time, an anomaly is declared. The value for timeout is either set to a default value equal to the deadline of the process associated with component port or can be specified by the system designer.
- *Data validity violation* (only applicable to consumers): Any event data token consumed by a consumer port has an associated expiration age. This is also known as the validity period in ARINC-653 sampling ports. We have extended this to be applicable to all types of component consumer ports, both periodic and aperiodic.
- *Pre-condition violation*: Developers can specify conditions that should be checked before executing. These conditions can be expressed over the current value or the historical change in the value, or rate of change of values of variables (with respect to previously known value for same parameter) such as
  1. the event data of asynchronous calls,
  2. function parameters of synchronous calls, and
  3. (monitored) state variables of the component.
- *User-code failure*: Any error or exception raised in the user code can be abstracted by the software developer as an error condition which can then be reported to the framework. Any unreported error is recognized as a potential unobservable discrepancy.
- *Post-condition violation*: Similar to pre-condition violations, but these conditions are checked after the execution of the function associated with the component port.
- *Deadline violation*: Any process execution must finish within the specified deadline.

These monitors can be specified via (1) attributes of model elements (e.g. Deadline, Data_Validity, Lock time out), and (2) via a simple expression language. The expressions can be formed over the (current) values of variables (parameters of the call, or state variables of the component), their *change* (delta) since the last invocation, their *rate* of change (change divided by a time interval). Table 1 provides the summary of anomalies that can be observed on a component port and the component as a whole. Code generators included in the design tools generate the appropriate code for the monitors. While most of the monitors described above are evaluated in the same thread executing the component port, the monitors associated with resource usage (i.e. CPU time) are run in parallel by framework. Figure 3 shows the flowchart of the code generated to handle incoming messages on a consumer port. The failed monitored discrepancy is always reported to the local component health manager. Deadline violation is always monitored in parallel by the runtime framework.

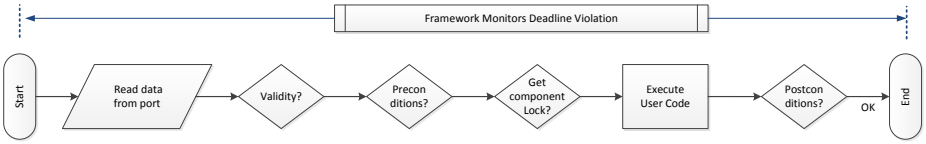


Fig. 3. Flow chart describing the interleaving of monitor and business logic provided by the user for a consumer port. The generated sequence is similar for other ports.

Table 1. Monitoring Specification. Comments are shown in italics.

<**Pre-condition**>::=<Condition>
<**Post-condition**>::=<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 timeout**>::=<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>::=&& | ||

Note 1. It is necessary to point out that the pre-conditions and post-conditions, if specified, should be verified against the formal system specification. We argue that it is easier to verify these conditions at run-time compared to formally verifying the full system. However, the full system should undergo rigorous testing[5]. During runtime, the formally verified conditions provide a blueprint for ensuring that the system/components are working without any discrepancy.

Example: In the GPS assembly shown in Figure 4, the ACM ports are configured with monitors of different kinds. Publisher and Consumer ports in Sensor, GPS, and NavDisplay are configured with monitors to track any violation of CPU resource usage (detected as Deadline Violation), the Publisher port in the Sensor component is configured to detect any violations/ problems with the user code (detected as User Code Violation), the Consumer ports in GPS and NavDisplay are configured to monitor problems with the age of the received data (detected as Data-Validity Violation), and Consumer and Receptacle port in NavDisplay are configured to detect Post-condition Violations.

[5] While formal verification covers all the possible behavior and environment interleaving traces, testing only covers the subset of all possible traces

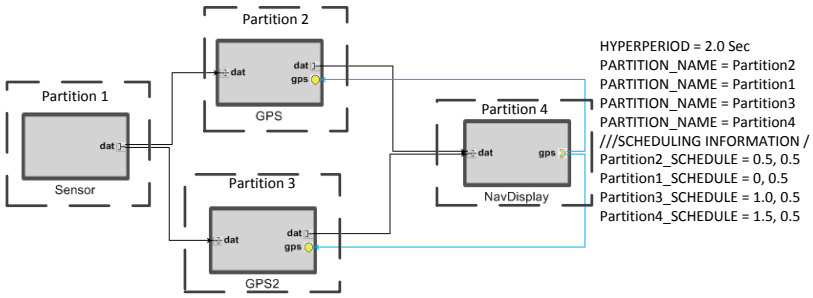


Fig. 4. Example: GPS Software Assembly. Unit of time is seconds.

3.3 System Integration

The modeling tools allow the system integrator to construct a system model by using the library of component models created by component developers. The modeling tool allows the system integrator to define the functionalities expected in the system and identify the appropriate components to provide these functionalities. The integrator creates the assembly model by instantiating and connecting the components, thereby capturing the interactions across the component assembly. At this time, the design constraints in the tools ensure that all ports are properly connected, e.g. the type of publisher matches the subscriber. The modeling tool also allows the system integrator to organize connected components into subsystems, which could then be reused to build more complex assemblies.

Once the assembly is specified logically the integrator can model the details of the platform and capture the deployment information. The modeling tool allows the specification of the platform in terms of the modules (i.e. processors) and the ARINC-653 partitions within each module. The integrator can specify the deployment of each component into an appropriate partition such that the temporal partitioning concerns are satisfied. At this time the integrators can use the integrated system model (assembly, platform, deployment models) to perform an end-to-end timing study on the system to check the logical correctness of design. Design tools are also used to fully generate the integration code and configuration files. These tools also generate the required build system along with necessary files to use the Eclipse IDE for final compilation and editing.

Example Figure 4 shows the integration model for the three GPS components showed in Figure 2. This model shows the connection between the components and their deployment on four different partitions. Partition 1 contains the Sensor Component. Partition 2 contains the GPS and Partition 4 contains the Navigation Display component. The sensor component publishes an event every 4 sec. The GPS component consumes the event published by sensor at a periodic rate of 4 sec. Afterwards it publishes an event, which is sporadically consumed by the

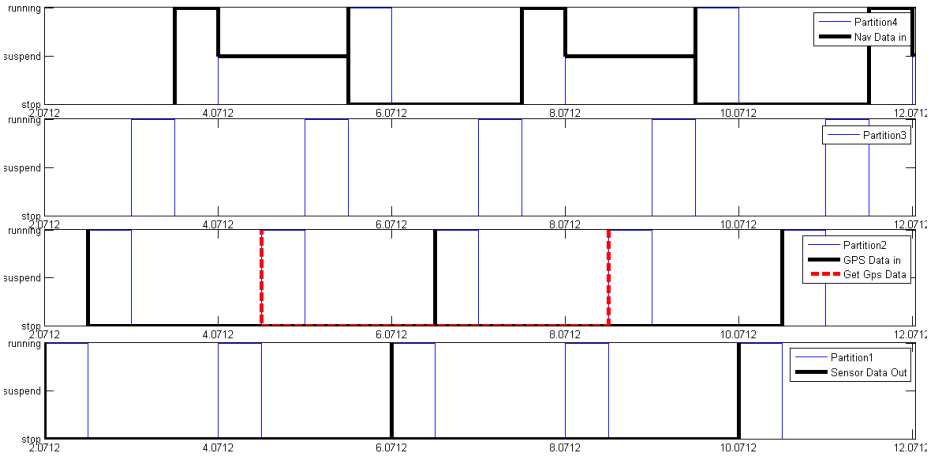


Fig. 5. Timing diagram for execution in the example.

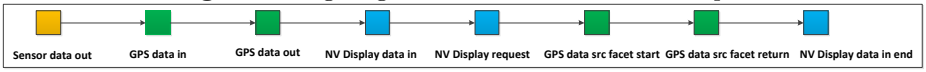


Fig. 6. The Chain of Events associated with data production and consumption across components in a hyper period.

Navigation Display (abbreviated as display). Thereafter, the display component updates its location by using getGPSData facet of the GPS Component. The publisher-consumer interaction between sensor and GPS components is implemented via a sampling port (Sampling ports are basic inter-partition communication mechanism in ARINC-653 platforms). A Channel connects the source sampling port from partition 1 to destination sampling port in partition 2. In this example, a redundant GPS is also connected in the assembly. The redundant component in this case shares the port structure with the other GPS. However, their internal behaviors are different. In this particular example, GPS 2 is set to the semi-active State i.e. it can consume but not publish.

Figure 4 also describes the periodic schedule followed by the partitions, overseen by a controller process called Module Manager [14]. This schedule is repeated every 2 s (hyper period). In each cycle, Partition 1 runs with a phase of 0 sec for 500 ms. (duration). Partition 2's phase is 500 ms. It runs for 500 ms. Then Partition 3 and Partition 4 run for next 1 second. This schedule ensures that the two partitions are temporally isolated. Figure 6 shows the timing diagram. Notice that the partitions are temporally isolated from each other. Ports are suspended when their partition is context switched. The GPS _data_in and Sensor_Data_out's execution time is very low. That is why they appear as impulses on the graph. The NavDisplay data in consumer takes longer to run because it invokes the Receptacle port to send a synchronous request to the GPS facet port, which cannot be fulfilled until GPS's partition becomes active.

Note 2. The temporal isolation and partition time allocation in this architecture is strict, i.e. activities in one partition do not affect activities in another partition unless there is an explicit data dependency. Even with data dependencies the time allocated to partitions remains as specified during design. Another point to note is that due to the use of model-driven tools and auto generation of the integration code, the system integrator can quickly change the deployment scenario by allocation each component to a different partition and regenerating the code. However, such a change requires the recompilation of affected partitions and can have an effect on the timing of the system.

4 Health Managers

In component-based systems, anomalies in a component can be either local or secondary effect of an anomaly in an upstream component. Identifying this pattern is important in order to isolate the root failure source. While the component level mitigation code (provided by a component developer) can quickly react to the local anomaly and possibly arrest any problems that could arise because of the anomaly, this mitigation action may not remove the primary source of failure. A system wide response/ mitigation engine would be ill-suited to react to every local anomaly but would be better positioned to identify and mitigate the real-fault source, especially when the failure effects cascade across component boundaries. Realizing the benefits and limitations of each strategy, we implemented a two level health management strategy in our framework with a *component level* that is local to a component, and the *system level* that covers the entire assembly of components. While the component level health manager is specified by the component developers, the system level health manager is provided by the system integrator. Both health managers are specified as hierarchical timed state machines using the modeling tools. Please refer to [15] for a formal description of these state machines.

Code generators are responsible for mapping the specified management logic to the runtime system, which ensures that the specified state machine logic is executed using a variation of the Harel state chart semantics [19]. Discussion of these semantics is not included in this paper. It should be noted that these managers are reactive because they are triggered by either an event, e.g. occurrence of an anomaly, or passage of time (i.e. a timeout). When triggered, the machine reevaluates its current state and in the process executes actions specified on the transition or for the state (entry, exit, or during). The next two sections describe both the component and system level health managers.

4.1 Component Level Health Manager

Component-Level Health Manager(CLHM) provides localized and limited functionality for managing the health of the internals of a component. The health manager reacts with appropriate mitigation action to the anomalies detected

Table 2. CLHM Mitigation Actions.

CLHM Action	**Semantics**
IGNORE()	Continue as if nothing has happened
ABORT()	Discontinue current operation, but operation can run again
USE_PAST _DATA()	Use most recent data (only for operations that expect fresh data)
STOP(p)	p is the process id. Default value is current process. This commands discontinues operation in process 'p'. Aperiodic processes (ports): operation can run again Periodic processes (ports): operation must be enabled by a future START HM action
START(p)	'p' is the same as defined in context of STOP (above). Re-enable a STOP-ped periodic operation
RESTART(p)	'p' is the same as defined in context of STOP (above). A Macro for STOP followed by a START for the current operation

within the component. As described in the previous section, CLHM is implemented as a timed state machine. It is triggered by anomalies detected by the monitors deployed inside the component, as shown in Table 1.

In addition to these monitors that detect and report anomalies, monitors to report *ENTRY* into and *EXIT* out of a port's process can also be specified using the modeling tool. These monitors aid in building observer models to track the execution sequence of component processes (ports) and report any deviations from the expected sequence. Observers are modeled as parallel state machines within the CLHM with one machine acting as an observer and another as the health manager. Each of the parallel state machines could be triggered by their relevant monitor events. While the observer tracks the state evolution, the health manager issues appropriate mitigation action for the anomalies detected. When an anomaly is detected in the observer, it triggers the health manager portion of the CLHM state machine to take the appropriate mitigation action.

The mitigation commands that can be expressed in the timed state machine model for CLHM are described in the Table 2. These commands can be issued as a transition action (executed when a state transition succeeds) or as entry, exit or during action of a state.

The CLHM associated with each component is hosted on a separate high priority ARINC-653 process. When a monitor reports a violation, the report is communicated to the relevant manager using the methods supported by the framework, e.g. an ARINC-653 buffer, see Figure 7. The buffer provides an intra-partition FIFO message queue for communication. This rerport triggers the execution of the CLHM state machine code which responds with an appropriate mitigation action. Depending on the nature of the mitigation action, the appropriate command is communicated either to the framework or to a relevant process which then executes it. Commands such as IGNORE, ABORT, USE_PAST_DATA are communicated to the managed process executing the monitor using a shared

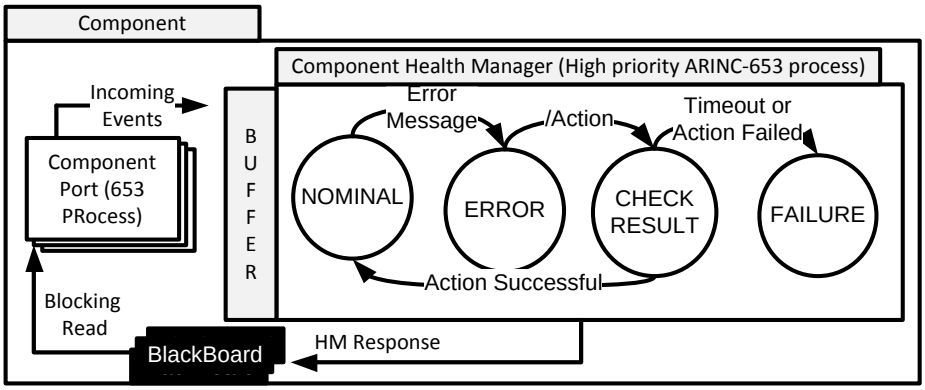


Fig. 7. Component Health Manager.

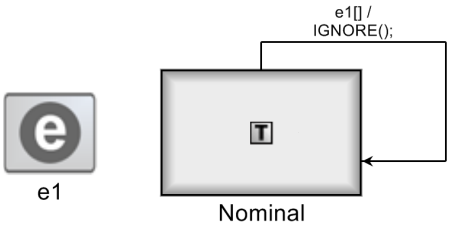


Fig. 8. Component Level Health Management Strategy for Sensor Component. Event e1 implies a user code exception in data out port.

memory resource called (a "blackboard" in [1]). START, STOP and RESTART commands are directly executed using the APIs of the framework.

Example Figure 8 shows the component health manager associated with the sensor component in the assembly shown in Figure 4. The timed-state machine specifying the CLHM for Sensor Component is triggered when a violation in the Publisher's (data_out) User-Code is detected. The monitor associated with detecting this violation is run on the same ARINC-653 process as the Publisher port. When the violation is detected, it is reported to the CLHM and the Publisher code blocks for a response/ command from the CLHM. The reported user code violation triggers the event e1 in CLHM state machine. In this case, the state-machine issues an IGNORE event which is translated as an IGNORE command and sent back to the Publisher port. Upon receipt of the command, the publisher port executes the command. In this case the IGNORE command results in the publisher continuing with its task (as per the semantics of the IGNORE command explained in the Table 2).

The timed-state machine associated with GPS CLHM is shown in Figure 9. This state machine is triggered whenever the input events e1 or e2 is triggered.

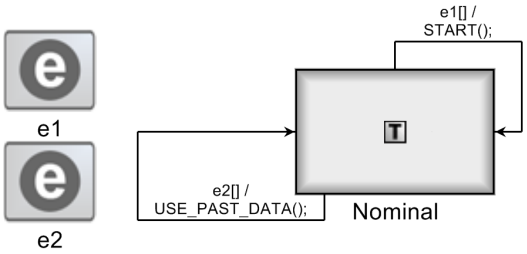


Fig. 9. Component Level Health Management Strategy for GPS Component. Event e1 implies deadline violation of data in port. Event e2 implies validity violation of data in port. Any other anomaly is sent the default IGNORE action.

The event e1 is triggered when a violation is detected in the resource usage (Deadline Violation) of the Consumer port (data_in). Event e2 is triggered when the age of the data-token received by the consumer port (data_in) is beyond its usable-time (Data-Validity Violation). The event e1 is triggered when the underlying framework detects a deadline violation and reports it to the CLHM. In this case, since the consumer port is configured with a HARD Deadline Type, the framework stops the process and then reports the violation to CLHM. This triggers the input event e1 in the state-machine. The state-machine executes the transition corresponding to the event e1 and issues a START event. This event results in a command to the framework to START the failed process associated with the consumer port. The event e2 in the CLHM state-machine is triggered when the Data-Validity violation is detected in a token received by the Consumer port. The Consumer port reports the same to the CLHM and blocks for a response from the CLHM. The CLHM executes the transition corresponding to the event e2 and triggers a USE_PAST_DATA output event which is sent as a USE_PAST_DATA command to the consumer port. The consumer port executes this command by replacing the current token with the past token and continues its operation. The Nav display machine is modeled in a similar fashion and is not shown here.

Scope of Component Level Health Manager Inputs (anomaly detected) and outputs (commands issued) of CLHM are local to a component. While this provides a quick fix to the detected problem which could prevent the effect of the problem from being propagated, it might not solve the root-cause of the problem. In the examples discussed above, it is quite possible that an anomaly detected in one component (e.g. validity violation in GPS) could have resulted from a problem in an upstream component (e.g. Sensor's Publisher user code that is responsible for data published). Also, an anomaly observed in one component could be the effect of a CLHM mitigation action executed in another component. A higher level health management unit is required to tackle the problem of fault

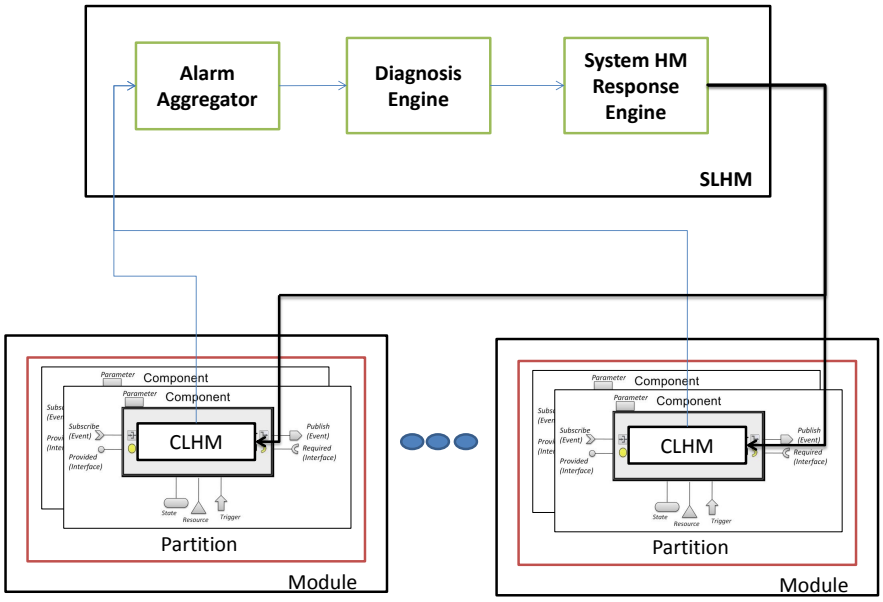


Fig. 10. SLHM Architecture. SLHM Components are automatically configured by the ACM design tools.

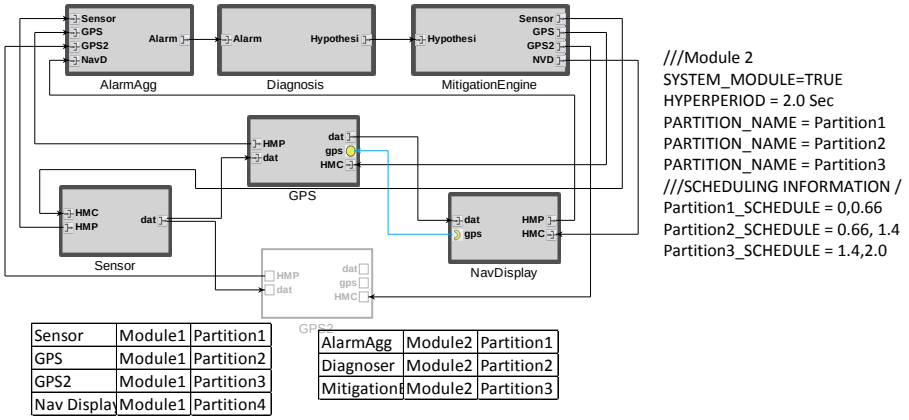
cascades across component boundaries. The next section deals with this second (or higher) level health management unit.

4.2 System-Level Health Manager

System Level Health Manager (SLHM), as the name suggests, is the health management strategy at the system-level. This section discusses in detail the enhancements that need to be made to the existing system made up of ACM components to enable System Level Health Management.

Architecting the Assembly model with the SLHM layer Architecting support for SLHM into the existing model involves adding special components that provide the core SLHM functionality and instrumenting the existing components in the assembly with the capability to exchange information with these special components. As shown in the Figure 10, the three special SLHM components include:

– *Alarm Aggregator*: It 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



Sensor	Module1	Partition1
GPS	Module1	Partition2
GPS2	Module1	Partition3
Nav Displa	Module1	Partition4

AlarmAgg	Module2	Partition1
Diagnoser	Module2	Partition2
Mitigation	Module2	Partition3

Fig. 11. GPS Assembly (ref, Figure 4) augmented with the SLHM components. This process is completely automated. The integrator only specified the internal of the response/ mitigation engine using as a timed state machine model and specifies the SLHM deployment.

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, whose default value is same as the hyperperiod, and sorts them based on their time of occurrence. A periodic publisher in the Alarm Aggregator feeds this sorted data to the Diagnosis Engine.

- *Diagnosis Engine*: It hosts an instance of a Timed Failure Propagation Graph reasoning engine. This engine is initialized by an auto-generated Timed Failure Propagation Graph (TFPG) model that captures the failure-modes, discrepancies and the failure propagation across the entire system. The reasoner uses this model to isolates the most plausible fault-source (component) that could explain the observations i.e. monitors triggered and the CLHM commands issued. The diagnosis result i.e. faulty component(s) is reported through an aperiodic publisher to the next component: the SystemHMResponse Engine that hosts the system level mitigation strategy.
- *SystemHM Response Engine*: It receives the diagnosis results: the set of faulty components and responds with an appropriate system-level command to mitigate the fault and its effects. This engine hosts a timed state-machine that executes the SLHM mitigation strategy specified by the user (described later in this section). The updated fault-status of the components in the assembly is used to trigger the SLHM state-machine. The output generated by the state machine is translated and sent (published) as mitigation commands to the appropriate components.

In order to enable communication between the existing component assembly and the SLHM layer, each components in the existing assembly is instrumented with an additional publisher (**HMPublisher**) and consumer (**HMConsumer**).

More specifically, the special publisher (**HMPublisher**) in these components feeds CLHM output (alarm detected and local mitigation action) to the Alarm Aggregator component. The special consumer (**HMConsumer**) in these components receives the mitigation command issued by the SystemHM Response Engine and executes it.

The modeling and generator support tools automatically update the design of the entire system with the special SLHM components, additional publisher and consumer in each of the existing components, and inter-connections between the components to capture the SLHM related information flow. Two additional pieces of information are required to complete the SLHM generation - the customized mitigation strategy to be executed by the SystemHM Response Engine and the deployment information for the three SLHM components. While the deployment information can be captured in a manner similar to the other (regular/functional) components in the assembly, the design tools allow the mitigation strategy to be specified as a state machine model. The code generators use the updated model to complete the generation and customization of the SLHM layer.

Example Figure 11 shows the GPS assembly described earlier in Figure 4 augmented with the three system health management components. Notice that each functional component i.e. Sensor, GPS, GPS2 and NavDisplay gets an additional publisher and Consumer. Anomaly/ alarms and mitigation commands are communicated through these ports. This process is completely automated. The integrator only specified the internal of the response/ mitigation engine using as a timed state machine model and specifies the SLHM deployment. In this particular example, SLHM components are deployed on a different processor (module) and divided into three partitions. This ensures that each stage of the SLHM gets a fixed time slice. The hyper period of this module is synchronized with the hyper period of the module containing the functional components of the GPS-Assembly. This ensures that system diagnosis, mitigation and transmission of message across the two modules run synchronously.

The following sections provide more detailed information with examples on the Diagnosis and Mitigation aspects of the SLHM layer.

4.3 Diagnosis : Isolation and Identification of the Fault Source

This section focuses in more detail on the diagnosis and mitigation aspects of system health manager. Our implementation of SLHM uses a reasoning scheme based on the Timed Failure Propagation Graph (TFPG) model[3,20].Timed failure propagation graphs (TFPG) are causal models that capture the temporal characteristics of failure propagation in dynamic systems. A TFPG is a labeled directed graph. Nodes in graph represent either failure modes (fault causes), or discrepancies (off-nominal conditions that are the effects of failure modes). Edges between nodes capture the propagation of the failure effect.

The TFPG model serves as the basis for a robust online diagnosis scheme that reasons about the system failures based on the events (alarms and modes)

observed in real-time[2,3,20]. The TFPG approach has been applied and evaluated for various aerospace and industrial systems[31].

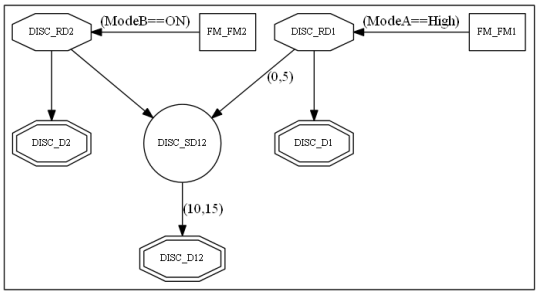


Fig. 12. An Illustrative TFPG Example. Doubled lined octagons are observed discrepancy. Single Line octagon are unobserved OR discrepancies. Unobserved AND discrepancy are denoted by circle. Rectangles are root failure nodes. Graph edge shows propagation link. Edge can be annotated with min and max propagation time. Absence of this annotation implies propagation delay lies within the interval $(0,\infty)$.

Example Figure 12 shows a simple non-hierarchical TFPG model. It shows the root causes of the failure (Failure-Modes FM_FM1, FM_FM2) and the anomalies (Discrepancies DISC_RD1, DISC_D1, DISC_SD12, DISC_D12, DISC_RD2, DISC_D2) that would be triggered when one or more of these failures occur. While failure modes are depicted as a box, unobserved OR-Discrepancies (e.g. DISC_RD1, DISC_RD2 etc.) are depicted as octagons, unobserved AND Discrepancies (DISC_SD12) are drawn as circles. Observable discrepancies (e.g. DISC_D1, DISC_D2, and DISC_D12) are drawn with a double-boundary. Edges in the graph capture the failure propagation starting from the Failure Modes to Discrepancies and then to subsequent Discrepancies downstream. Some of these links depict additional constraints related to activation-condition and timing for failure propagation. The activation condition (a Boolean expression over the modes) captures when failure can propagate over a link. The timing constraint expresses the time bounds within which the failure effect is expected to propagate over that link. When these constraints are absent, the failure can propagate over the link at any time or in any mode.

4.4 Automated Synthesis of TFPG from ACM Assembly Model

The information present in the ACM assembly model allows us to automatically synthesize the TFPG model of the system. This TFPG model is built on a hierarchical basis. Initially the TFPG models of the component ports are created. These component port TFPG models are then used to build the TFPG models

of the Components which are then used to build the TFPG model of the entire Assembly.

The TFPG-model for each component-port type is constructed using the knowledge of the sequence of operation (for each port-type) and the fault-sources and anomalies associated with each operation in the sequence. The TFPG model links the fault-sources/ failure-modes and the anomalies/ discrepancies (monitored/ unmonitored) across the sequence of operations. It also contains input and output discrepancies that represent anomalies that propagate in or propagate out of the port.

The TFPG model of the component is then constructed by instantiating the appropriate component-port TFPG model for each component port present in the component. TFPG model of the component includes additional failure modes and anomalies specific to the component. The Component TFPG model is completed by adding the failure propagation links between the fault-sources and anomalies present in the component and its ports. This is done by using the data and control flow information captured in the models of the software components.

The TFPG model of an assembly is constructed using the TFPG models of the components present in the assembly. The failure propagation links between the component TFPG models are added on the basis of the integration information i.e. inter-component interaction information (publisher-consumer, facet-receptacle) present in the assembly.

The activation conditions for the failure propagation links in a component-port are expressed in terms of the mitigation commands issued by the CLHM e.g. An IGNORE command from the CLHM could imply that the failure could propagate and trigger anomalies downstream e.g. an invalid data being published or an invalid state update. An ABORT command from the CLHM could stop the failure propagation, but it also stops the normal sequence of operation of the port, thereby leading to no data being published or no update to the state. The failure propagation links across the component boundaries have activation conditions that are based on the states of the two associated components: active, inactive, semi-active. A detailed discussion of the TFPG templates associated with the port is not included in this chapter. Interested readers are referred to Appendix D in [13] and the example in [27].

4.5 Example

Figure 13 shows a portion of the TFPG model of the entire GPS Assembly. It shows the TFPG model of the sensor component, the TFPG for GPS data_in port, and the failure propagation between them.

The TFPG models of ports of components have a failure mode: FM_USER_CODE. This failure mode arises from the latent bug in implementation code. Both TFPG models also contain anomalies associated with violations observed in the normal sequence of operation: data validity, pre-condition, user code exceptions, post-conditions and deadline violations. Causal relation across these anomalies is dictated by the operation sequence in the port. For example, an

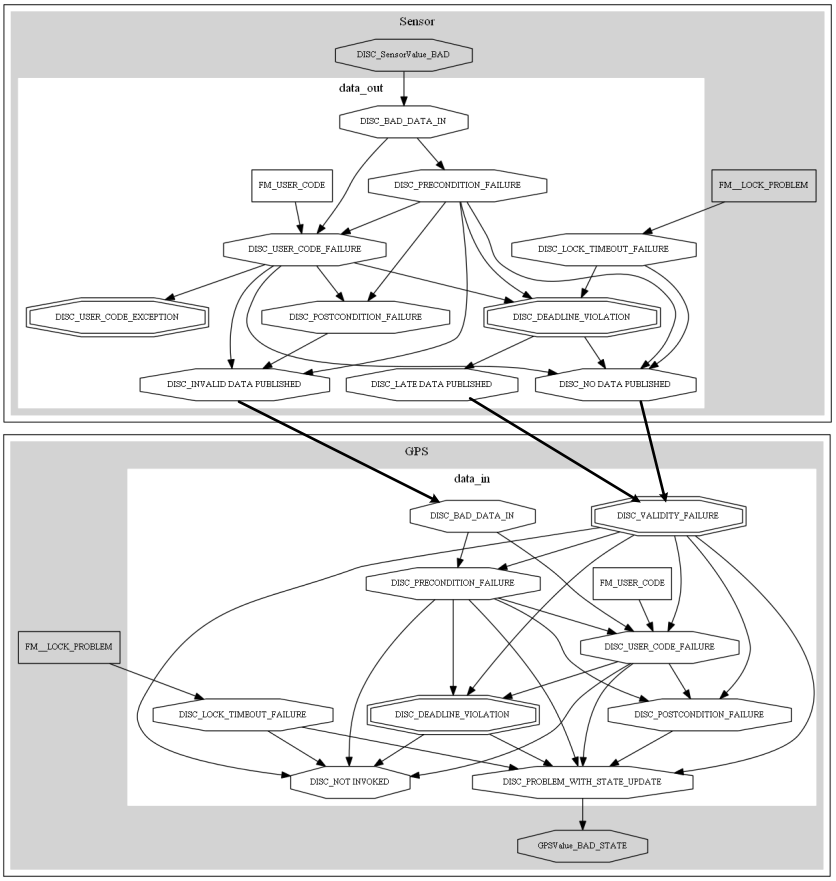


Fig. 13. TFPG model for Sensor-Publisher and GPS-Consumer

anomaly in the pre-condition could lead to an anomaly in the user code, which again could lead to a post-condition or deadline-violation. Latent bugs (i.e. FM_USER_CODE) can lead to an exception in the user code or can lead to deadline-violation or post-condition violation.

As stated earlier, port TFPG models here also contain input and output discrepancies that represent anomalies that propagate in or propagate out of the port. In Figure 13, DISC_BAD_DATA_IN is associated with bad-data getting into a port from a state variable (in case of a sensor-publisher) or a publisher (in case of a GPS-consumer). DISC_LOCK_TIMEOUT_FAILURE represents another input-discrepancy that is associated with inability to secure the component-lock. Anomalies propagating out of the publisher port are captured by discrepancies associated with the published data, e.g. DISC_NO_DATA_PUBLISHED, DISC_LATE_DATA_PUBLISHED, and DISC_INVALID_DATA_PUBLISHED. Similarly, anomalies out of consumer port are captured by discrepancies

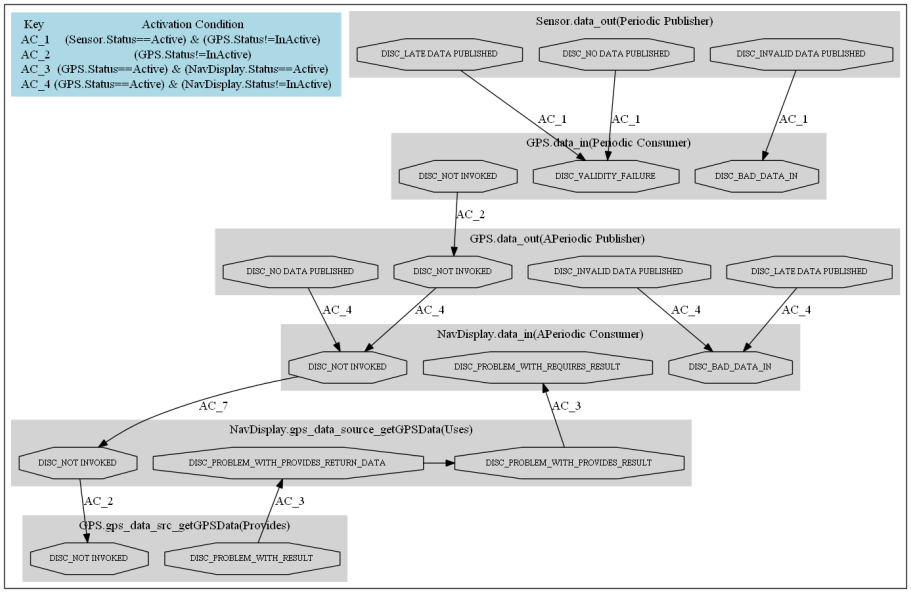


Fig. 14. Intra-component and Inter-component Failure Propagation associated with the control-flow in the GPS Assembly

that are associated with problem in the state-update , e.g. DISC_PROBLEM_-WITH_ STATE_UPDATE.

Activation conditions on failure propagation links are not shown in Figure 13. These conditions are based on local mitigation commands. For example, an IGNORE or USE_PAST_DATA command from CLHM in response to a violation detected in a pre-condition can cause problems in the user code or post-condition, or deadline violation. Finally, this could lead to a bad output data. On the other hand, an ABORT command can arrest the failure propagation in nominal operation sequence but could introduce other effects such as no output data (e.g DISC_NO_DATA_PUBLISHED).

Figure 13, also shows the component wide failure modes such as FM_LOCK-_PROBLEM. This failure mode represents the problem associated with synchronization among component ports. The effect of this failure manifests in a component port through the discrepancy: DISC_LOCK_TIMEOUT_FAILURE. Component TFPG models also contain anomalies associated with bad values in the state variables: DISC_Sensor_Bad_Value and GPSValue_Bad_State. These anomalies help in capturing the failure propagation based on the dataflow model of the component. The bad data produced in a port could lead to a bad state variable update in a component. If the state variable is being used by a publisher port for publishing data, the bad state variable update can lead to an invalid data being published from the publisher port. This failure propagation associated with dataflow is not restricted to the component boundary.

Dataflow due to the component port interactions captured in the assembly model could lead to failure propagations across component boundaries. Figure 13 captures these failure propagations across component boundaries between the sensor' publisher port to the GPS's consumer port (dark edges in the Figure 13). These failure propagations capture the effect of problems in the sensor's publisher trickling down into GPS's consumer leading to a bad input data (DISC_BAD_DATA_IN) or a data validity violation (DISC_DATA_VALIDITY).

Failure effects are also propagated along the control flow, e.g. when a port is not invoked: DISC_NOT_INVOKED. This happens as the control flow is disrupted when the normal sequence of operation is affected in a port. This can happen across component boundary if a facet-receptacle interaction exists or within the component when a port is responsible for invoking another port, e.g. a periodic consumer can invoke an aperiodic publisher. In Figure 14, this relationship exists between GPS data_in and GPS data_out. A lack of invocation of a port can affect the state update inside the component.

Figure 14 captures the explicit failure propagations across component port boundaries. These failure propagations include those introduced by dataflow as well as control flow. To avoid clutter, the Figure 14 restricts the depiction to anomalies within component ports that are associated with direct failure propagations across component-port or component boundaries. Other failure modes, anomalies, and failure propagations within component port boundaries are not shown.

4.6 System Level Diagnosis Process

The TFPG diagnosis engine hosted inside the SHM component is instantiated with the generated TFPG model of the system/assembly. When it receives the first alarm from a fault scenario, it reasons about it by generating all hypotheses for failure modes that could have possibly triggered the alarm. Each hypothesis lists its possible failure modes and their possible timing interval, the triggered alarms that are supportive of the hypothesis, the triggered alarms that are inconsistent with the hypothesis, the missing alarms that should have triggered, and the alarms that are expected to trigger in future. Additionally, the reasoner computes hypothesis metrics such as plausibility and robustness that provide a means of comparison. These metrics are used to prune the hypotheses set such that only those hypotheses with a higher metric and hence better explanation are retained [2]. At this time only hypothesis with 100 percent plausibility are used for failure mitigation. Output of diagnosis engine i.e. the hypothesis of failed component is sent to the timed state machine implementing the System Level Mitigation Strategy.

Example Consider a failure scenario such that there is a user code bug in the sensor data_out process such that the code does not publish the data in its time duration. Since sensor data out process does not have a monitored post-condition discrepancy (see Figure 13) there will be no alarms generated in the

Table 3. SLHM Functions. Here c denotes the component name and s denotes a subsystem name. Unless otherwise specified usage of the subsystem name in a command implies apply to all contained components.

Action	**Semantics**
IS_FAULTY (c|s)	Returns true if the component is faulty.. A subsystem is marked as faulty if the minimum number of components required for work is not available.
IS_NOT_FAULTY (c)	Returns false if the component is faulty.[1]
RESET (c|s)	Instructs the component to execute its Reset method.
STOP (c|s)	Instructs the component to switch to inactive mode. Component stops executing the functionality of all its ports. If subsystem is argument, command is applied to all its components
START (c|s)	Instructs the component to switch to active mode. Component starts executing the functionality of all its ports.
DISABLE_OUTPUT (c|s)	Instructs the component to switch to semi-active mode. Only Consumer and Receptacle ports are operational.
REWIRE (c,i,pc)	i: Interface Name, pc: Provider Component Name. This command Instructs Component (c) to switch its receptacle Interface (i) to connect to the appropriate facet interface in another component (pc).
CHECKPOINT (c|s)	Instructs the component to Checkpoint its current state-variables.
RESTORE (c|s)	Instructs the component to Restore its state-variables from the Checkpoint.

[1] A corresponding method can be implemented for the subsystem and used in a specific example. Currently, implementation of this method is system specific and is not part of provided API.

sensor process. However, due to the user code problem, the silent (unobserved) discrepancy user code failure will be triggered, which will then lead to either silent post-condition failure, or late data published, or no data published.

Now, consider the GPS data in process in the same figure. In this process, a validity violation will be raised as no data is being received from the publisher. This will cause the local health manager to issue a USE_PAST_DATA command (Figure 9). The raised alarm of validity violation along with the USE_PAST_DATA command will be reported to the diagnosis engine. Inside the diagnosis engine, the event of 'validity violation' will be used to produce the most plausible hypothesis (root failure sources) that can explain the observed anomaly. In this particular case the TFPG engine will correctly attribute the problem to either Sensor Lock failure mode or Sensor user code failure mode, i.e. a faulty sensor component.

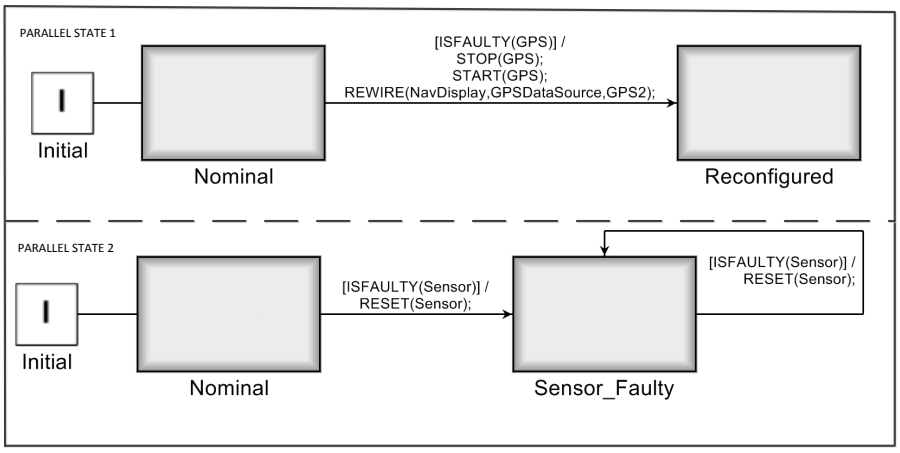


Fig. 15. System Level Health Manager Mitigation Strategy for GPS Assembly

4.7 System Level Mitigation Strategy

The system level mitigation strategy is also modeled as a hierarchical timed state machine. Table 3 lists the statements (functions) that can be used in the state machine to express the guard conditions (to check if a component is faulty) and actions (i.e. mitigation commands). These strategies are reactive in nature and aim to restore the functionality by cold/warm reset or switching to redundant component. As mentioned earlier in this chapter, each component in the assembly is assumed to be in one of the three possible states: inactive, active, and semi-active. When the component is in inactive state, none of the ports in the component perform their task. The active state of a component is the exact opposite to inactive state, and all the component ports performing their task. In a semi-active state, only the consumer and receptacle ports of a component are operational. The publisher and provided ports are disabled. This state-machine is translated into operational code and is hosted inside the runtime of the System Level Health Management module.

An alternative strategy of health management is to search over available solutions to find the best option that can ensure that the system functionalities are still met. This strategy is still under active investigation.

Example Figure 15 shows the state-machine model of the System Level Mitigation Strategy associated with the GPS Assembly. In this case the mitigation strategy involves parallel state machines that deal with problems associated with Sensor component (parallel-state 1) and GPS component (parallel-state 2). The top level state machine is triggered when there is updated diagnosis information (hypothesis) from the diagnosis engine. This information is used by the System response/ mitigation engine to update the list of faulty components and trigger

the SLHM state machine. When the SLHM state machine (Figure 15) associated with the GPS Assembly is invoked it triggers Parallel State-1 followed by Parallel State-2.

In Parallel-State 1, the System Health Manager checks if the GPS component is marked as faulty. This is captured in the transition guard condition: IS_FAULTY(GPS). If this guard condition evaluates to true, then the transition action statements are executed and the state changed. The transition action statements direct the reconfiguration to the alternate GPS (GPS2) through a series of statements. STOP(GPS) results in a STOP command being sent to the GPS component. START(GPS2) results in a START command being sent to the GPS2 component. The command REWIRE(NavDisplay,GPSDataSource, GPS2) directs a rewire command to the NavDisplay component. It instructs the component to rewire the receptacle interface (GPSDataSource) to the appropriate facet in the component GPS2.

In Parallel-State 2 it checks if the Sensor component has been marked as faulty. This is captured in the transition guard condition: IS_FAULTY(Sensor). If this guard condition evaluates to true, then an output event is triggered to reset the sensor component (transition action: RESET(Sensor)). This translates into a RESET command that is sent to the Sensor component. The Sensor component then executes the Reset method associated with the component and reports back to the system health manager.

An additional case study of an Inertial Measurement Unit (IMU)System built using the ACM design tools is available as a tech report for interested readers [16].

5 Known Limitations and Future Work

While the results of the experiments indicate that the approach is feasible and very general, and shows the promise of being able to scale to and handle real-life problems, we do understand that architecting a software health management system is contingent upon the availability of extra resources that can be spared for this purpose. This implies the necessity of scheduling analysis that considers the future state change of components as part of system mitigation. On the modeling front, the current state-machine based mitigation strategy requires explicit specification of the mitigation action for each fault in the system. This might become unwieldy beyond a point. We are focusing on alternate strategies to specify the mitigation action. We are exploring the use of function-allocation models in conjunction with automated reasoning strategies to tackle the mitigation problem by identify and switching to available redundancies to restore the affected functionality. Further, we need to explore effective means to use the diagnosis result when it is less than perfect i.e. when the hypotheses are not good enough to accurately determine the faulty component. On the diagnosis front, it would be ideal if all the possible monitors (i.e. pre-conditions, post-conditions, invariants) are configured and available for use with the TFPG diagnosis model. In an ideal monitoring situation where all monitors are configured correctly and

fire in the correct sequence, this will help prune the hypotheses set faster and come up with a quick and correct diagnosis result. However, we do understand that it might not be possible to configure all the available monitors and more importantly and in some cases these monitors could not be reliable. We plan to work on strategies where less than perfect diagnosis results (lots of ambiguities and/or lack of hypotheses that have hundred percent plausibility) can be effectively handled to restore the system functionality.

6 Conclusion

In summary, the paper describes a technology for implementing fault adaptivity in real-time systems using as software health management approach. The starting point of the technology is a real-time component model that introduces component-based software engineering techniques into real-time systems. Components, their interfaces, and interactions are explicitly modeled, and these models are annotated with observable pre- and post-conditions, as well as timing requirements. An anomaly detection system is constructed from these specifications. It performs the monitoring on the software system, and, if needed, triggers a health management (mitigation) action. Health management can happen on the component or on the system-level: in the first case the mitigation is facilitated by a designer-specified reactive state machine, in the second case a diagnosis process is triggered first, whose results are then used in a reactive or deliberative response/ mitigation engine. The diagnosis is necessary to isolate source of cascading faults that propagate through multiple components. We have built a model-driven tool chain for developing these systems, and we have evaluated the approach on several laboratory examples and demonstrated the effectiveness of the concept on some large ones that replicate real-life incidents.

References

1. ARINC specification 653-2: Avionics application software standard interface part 1 - required services. Tech. rep.
2. Abdelwahed, S., Karsai, G., Mahadevan, N., Ofsthun, S.C.: Practical considerations in systems diagnosis using timed failure propagation graph models. Instrumentation and Measurement, IEEE Transactions on 58(2), 240–247 (February 2009)
3. Abdelwahed, S., Karsai, G., Biswas, G.: A consistency-based robust diagnosis approach for temporal causal systems. In: 16th International Workshop on Principles of Diagnosis. pp. 73–79 (2005)
4. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (Jan 2004)
5. Bureau, A.T.S.: In-flight upset; 240km NW Perth, WA; Boeing Co 777-200, 9M-MRG. Tech. rep. (August 2005), `http://www.atsb.gov.au/publications/investigation_reports/2005/AAIR/aair200503722.aspx`

6. Bureau, A.T.S.: AO-2008-070: In-flight upset, 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. Tech. rep. (October 2008), http://www.atsb.gov.au/publications/investigation_reports/2008/AAIR/aair200806143.aspx
7. Bustard, D.W., Sterritt, R.: A requirements engineering perspective on autonomic systems development. Autonomic Computing: Concepts, Infrastructure, and Applications pp. 19–33 (2006)
8. Butler, R.: A primer on architectural level fault tolerance. Tech. rep., NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108 (2008), http://shemesh.larc.nasa.gov/fm/papers/Butler-TM-2008-215108-Primer-FT.pdf
9. Charette, R.: This car runs on code. IEEE Spectrum, February (2009)
10. Cheng, B.H.: Software engineering for self-adaptive systems. chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02161-9_1
11. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. pp. 21–26. ACM Press, New York, NY, USA (2002)
12. DO-178B, Software considerations in airborne systems and equipment certification. RTCA, Incorporated (1992)
13. Dubey, A., Karsai, G., Mahadevan, N.: Towards model-based software health management for real-time systems. Tech. Rep. ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University (August 2010), http://isis.vanderbilt.edu/node/4196
14. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: CCM with ARINC-653. Software: Practice and Experience 41(12), 1517–1550 (2011), http://dx.doi.org/10.1002/spe.1083
15. Dubey, A., Karsai, G., Mahadevan, N.: Model-based Software Health Management for Real-Time Systems. In: Aerospace Conference, 2011 IEEE. pp. 1–18. IEEE (2011)
16. Dubey, A., Mahadevan, N., Karsai, G.: The inertial measurement unit example: A software health management case study. Tech. Rep. ISIS-12-101, Institute for Software Integrated Systems, Vanderbilt University (February 2012), http://isis.vanderbilt.edu/node/4496
17. Garlan, D., Cheng, S.W., Schmerl, B.: Architecting dependable systems. chap. Increasing system dependability through architecture-based self-repair, pp. 61–89. Springer-Verlag, Berlin, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=1768179.1768183
18. Greenwell, W.S., Knight, J., Knight, J.C.: What should aviation safety incidents teach us? In: SAFECOMP 2003, The 22nd International Conference on Computer Safety, Reliability and Security (2003)
19. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8(3), 231 – 274 (1987), http://www.sciencedirect.com/science/article/pii/0167642387900359
20. Hayden, S., Oza, N., Mah, R., Mackey, R., Narasimhan, S., Karsai, G., Poll, S., Deb, S., Shirley, M.: Diagnostic technology evaluation report for on-board crew launch vehicle. Tech. rep., NASA (2006)
21. Jaffe, M., Busser, R., Daniels, D., Delseny, H., Romanski, G.: Progress report on some proposed upgrades to the conceptual underpinnings of do-178b/ed-12b. In: System Safety, 2008 3rd IET International Conference on. pp. 1–6. IET (2008)

22. Johnson, S., Gormley, T., Kessler, S., Mott, C., Patterson-Hine, A., Reichard, K., Scandura Jr, P.: System Health Management: With Aerospace Applications. John Wiley & Sons, Inc (2011)
23. de Lemos, R.: Analysing failure behaviours in component interaction. Journal of Systems and Software 71(1-2), 97 – 115 (2004)
24. Lightstone, S.: Seven software engineering principles for autonomic computing development. ISSE 3(1), 71–74 (2007)
25. Lyu, M.R.: Software Fault Tolerance, vol. New York, NY, USA. John Wiley & Sons, Inc (1995), http://www.cse.cuhk.edu.hk/~lyu/book/sft/
26. Lyu, M.R.: Software reliability engineering: A roadmap. In: 2007 Future of Software Engineering. pp. 153–170. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/FOSE.2007.24
27. Mahadevan, N., Dubey, A., Karsai, G.: Application of software health management techniques. In: Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '11, ACM, ACM, New York, NY, USA (2011)
28. Potocti de Montalk, J.: Computer software in civil aircraft. In: Digital Avionics Systems Conference, 1991. Proceedings., IEEE/AIAA 10th. pp. 324 –330 (oct 1991)
29. NASA: Report on the loss of the mars polar lander and deep space 2 missions. Tech. rep., NASA (2000), ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/2000_mpl_report_1.pdf
30. Ofsthun, S.: Integrated vehicle health management for aerospace platforms. Instrumentation Measurement Magazine, IEEE 5(3), 21 – 24 (Sep 2002)
31. Ofsthun, S.C., Abdelwahed, S.: Practical applications of timed failure propagation graphs for vehicle diagnosis. In: Proc. IEEE Autotestcon. pp. 250–259 (17–20 Sept 2007)
32. Prisaznuk, P.: Arinc 653 role in integrated modular avionics (IMA). In: Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th. pp. 1.E.5–1 – 1.E.5–10. IEEE (2008)
33. Pullum, L.L.: Software fault tolerance techniques and implementation. Artech House, Inc., Norwood, MA, USA (2001)
34. Robertson, P., Williams, B.: Automatic recovery from software failure. Commun. ACM 49(3), 41–47 (2006)
35. Rohr, M., Boskovic, M., Giesecke, S., Hasselbring, W.: Model-driven development of self-managing software systems. In: Proceedings of the Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS/UML'06) (2006)
36. Sha, L.: The complexity challenge in modern avionics software. In: National Workshop on Aviation Software Systems: Design for Certifiably Dependable Systems (2006)
37. Shaw, M.: "self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. pp. 111–114. ACM Press, New York, NY, USA (2002)
38. Srivastava, A., Schumann, J.: The Case for Software Health Management. In: Fourth IEEE International Conference on Space Mission Challenges for Information Technology, 2011. SMC-IT 2011. pp. 3–9 (August 2011)
39. Taleb-Bendiab, A., Bustard, D.W., Sterritt, R., Laws, A.G., Keenan, F.: Model-based self-managing systems engineering. In: DEXA Workshops. pp. 155–159 (2005)

40. Torres-Pomales, W.: Software fault tolerance: A tutorial. Tech. rep., NASA (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307
41. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electron. Notes Theor. Comput. Sci. 141(3), 53–71 (2005)
42. Wang, N., Schmidt, D.C., O'Ryan, C.: Overview of the CORBA component model. Component-based software engineering: putting the pieces together pp. 557–571 (2001)
43. Williams, B., Williams, B., Ingham, M., Chung, S., Elliott, P.: Model-based programming of intelligent embedded systems and robotic space explorers. Proceedings of the IEEE 91(1), 212–237 (2003)
44. Williams, B.C., Ingham, M., Chung, S., Elliott, P., Hofbaur, M., Sullivan, G.T.: Model-based programming of fault-aware systems. AI Magazine 24(4), 61–75 (2004)
45. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. In: WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems. pp. 1–7. ACM, New York, NY, USA (2005)
46. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE '06: Proceeding of the 28th international conference on Software engineering. pp. 371–380. ACM, New York, NY, USA (2006)