Model-based Software Health Management for Real-Time Systems

Abhishek Dubey Gabor Karsai Nagabhushan Mahadevan
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN

Abstract—Complexity of software systems has reached the point where we need run-time mechanisms that can be used to provide fault management services. Testing and verification may not cover all possible scenarios that a system will encounter, 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 systems and implements a two level health management strategy that can be applied through a model-based software development process. The Component-level Health Manager (CLHM) for software components provides a localized and limited functionality for managing the health of a component locally. It also reports to the higher-level System Health Manager (SHM) which manages the health of the overall system. SHM consists of a diagnosis engine that uses the timed fault propagation (TFPG) model based on the component assembly. It reasons about the anomalies reported by CLHM and hypothesizes about the possible fault sources. Thereafter, necessary system level mitigation action can be taken. System-level mitigation approaches are subject of ongoing investigations and have not been included in this paper. We conclude the paper with case study and discussion.

TABLE OF CONTENTS

1. INTRODUCTION

Core logic for functions in complex cyber-physical systems like aircraft and automobiles is increasingly being implemented in software. Software was originally used to implement subsystem-specific functions (e.g. an anti-lock braking system in cars), but today software interacts with other subsystems as well e.g. with the engine control or the vehicle stability system and is responsible for their coordinated operation. It is self-evident that the correctness of software is essential for overall system functions.

As the complexity of software increases, existing verification and testing technology can barely keep up. Novel methods based on formal (mathematical) techniques are being used for verifying critical software functions, but less critical software systems are often not subjected to the same rigorous verification. There is a high likelihood for defects in software that manifest themselves only under exceptional circumstances. These circumstances may include faults in the hardware system, including both the computing and non-computing hardware. Often, the system is not prepared for such faults.

There is a well-established literature of software fault tolerance wherein some of the techniques of hardware fault tolerance based on redundancy and voting, like triple modular redundancy, are applied to the software domain [19], [27], [8]. While the architectural principles of software fault tolerance are clear, the complexity of software and various interconnections has grown to the point that by itself this has become a potential source of faults; i.e. the implementation of software fault tolerance may lead to faults. We argue therefore that such techniques do not provide a sufficient technology anymore and additional approaches are needed.

The answer, arguably, lies in two principles: (1) the software fault management should be kept as simple as possible, and (2) the software fault management system should be built according to very strict standards - possibly automatically generated from specifications. We conjecture that these goals can be achieved if software fault management technology embraces new software development paradigms, like

[1] 978-1-4244-7351-9/11/$26.00 ©2011 IEEE.
[2] IEEEAC Paper #1650, Version 2, Updated 12/28/2010.

component-based software and model-driven development.

Furthermore, current software fault management can be enhanced by borrowing additional techniques from the field of system health management that deals with complex engineering systems where faults in their operation must be detected, diagnosed, mitigated, and prognosticated. System health management typically includes the activities of anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline) [23], [18]. The techniques of SHM are typically mathematical algorithms and engineering processes, possibly implemented on some computational system that provides health management functions for the operator, for the maintainer, and for the sustaining engineer.

Some points to note about system health management and typical software fault tolerant design are: (1) system health management deals with the entire system, not only with a single subsystem or component; which is typically the case in software fault-tolerance approaches, (2) while fault tolerance primarily deals with abrupt, catastrophic faults, system health management operates in continuum ranging from simple anomalies through degradations to abrupt and complete faults, and (3) while the goal of typical software fault tolerance techniques is to mask the failure, health management explicitly aims at isolating the root failure and even predicting future faults from early precursor anomalies of those faults.[3]

In this paper we discuss the principles of software health management, in a model-based conceptual and development framework. First we discuss the model-based approach we follow, then explain a software component model we developed, show how the model can serve for constructing component level and system level health management services, and then illustrate its use through a case study. The paper concludes with a brief review of the related work and a summary.

2. Background on Model-Based Design

In the past 15 years a novel approach to the development of complex software systems has been developed and applied: model-driven development (MDD). The key idea is to use models in all phases of the development: analysis, design, implementation, testing, maintenance and evolution. This approach has been codified in two related and overlapping directions: the Model-driven Architecture (MDA) [3] of the Object Management Group (OMG), and the Model-Integrated Computing (MIC) [4] approach that our team advocates. MDD relies on the use of models that capture relevant properties of the system to be developed (e.g. requirements, architecture, behaviors, components, etc.) and uses these models in generating (or modifying) code, other engineering artifacts, etc. Perhaps the greatest success of MDD is in the field of embedded control systems and signal processing: today's flight software is often developed in Simulink/Stateflow [2] or Matrix-X [5] - that implement their own flavor of MDD. Properties of MDD relevant for the goals of software health management are as follows:

1. Models represent the system, its requirements, its components and their behaviors, and these models capture the designer's knowledge of the system.
2. Models are, in essence, higher-level programs that influence many details of the implementation.
3. Models could be available at operation time, e.g. embedded in the running system.
4. For this study, the system built using MDD is component-based: software is decomposed into well-defined components that are executed under the control of a component platform - a sort of 'operating system' for components that provides services for coordinating component interactions.
5. The component architecture is clearly reflected in and explicitly modeled by the models.

In the MDA approach, the key notion is the use of Platform-Independent Models (PIMs) to describe the system in high-level terms, then refine these models (possibly using model transformations) into Platform-Specific Models (PSMs) which are then directly used in the implementation (which itself could - wholly or partially - be generated from models). In the MIC approach, the use of Domain-Specific Modeling Languages (DSMLs) is advocated (that allow increases in productivity via the use of domain-specific abstractions), as well as the application of model transformations for integrating analysis and other tools into an MDD process. In either case, the central notion is that of the model, which is tightly coupled to the actual implementation, and the implementation (code) cannot exists without it.

3. Principles of Software Health Management

Health management is performed on the running system with the goal to diagnose and isolate faults close to their source so that a fault in a sub-system does not lead to a general failure of the global system. It involves four different phases:

1. *Detection*: Anomalous behavior is detected by observing various measurements. Typically, an anomaly constitutes violation of certain conditions which should be satisfied by the system or the sub-system.
2. *Isolation*: Having detected one or more anomalies, the goal is to isolate the potential source(s) of fault(s);
3. *Mitigation*: Given the current system state and the isolated fault source(s), mitigation implies taking actions to reduce or eliminate the fault effects;
4. *Prognostics*: Looking forward in time, prognostics is done to predict future observable anomalies, faults, and failures.

To apply these techniques to software we must start by identifying the basic 'Fault Containment Units'. We assume that *software systems are built from 'software components'*, where

[3] This is also true for Byzantine failures. While voting techniques can mask byzantine failure, a holistic system-wide approach is required for isolating the root failure mode and taking necessary actions.

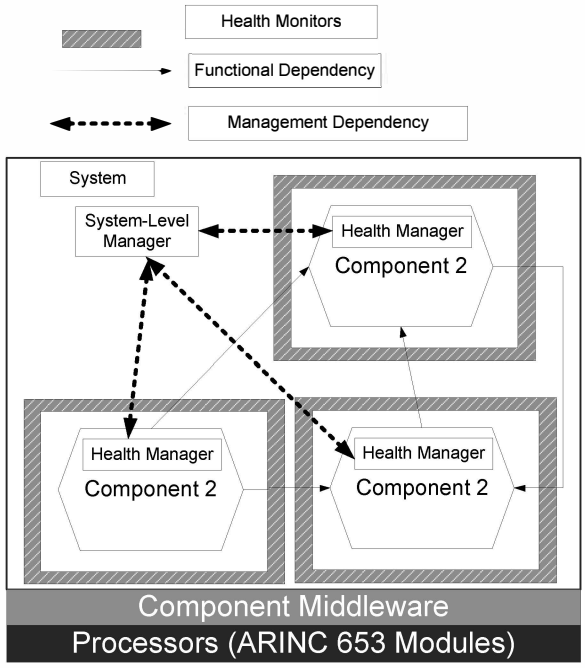


Figure 1. Hierarchical Layout of Component-Level and System-Level Health Managers

each component is a fault containment unit. Components encapsulate (and generalize) objects that provide functionality and we expect that these components are well-defined, independently developed, verified, and tested. Furthermore, all communication and synchronization among components is facilitated by a component framework that provides services for all component interactions, and no component interactions happen through 'out-of-band' channels. This component framework acts as a middleware, provides composition services, and facilitates all messaging and synchronization among components, and is used to support fault management.

Section 4 provides a brief background on the component framework used for the work presented in this paper. This framework assumes that the underlying operating system is ARINC-653 [1] compliant, state of the art operating system used in Integrated Modular Avionics. Appendix A provides a brief overview of ARINC-653[4].

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. As shown in figure 1, we have focused on two levels of software health management: *Component level* that is limited to the component, and the *System level* that includes system level information for doing diagnosis to identify the root failure mode(s).

[4]Please note that even though this paper uses an ARINC-653 based framework, these techniques are generic and can be applied to other real-time systems that can be configured statically during initialization.

Component-level health management (CLHM) for software components detects anomalies, identifies and isolates the fault causes of those anomalies (if feasible), prognosticates future faults, and mitigates effects of faults – on the level of individual components. We envision CLHM implemented as a 'side-by-side' object that is attached to a specific component and acts as its health manager. It provides a localized and limited functionality for managing the health of one component, but it also reports to higher-level health manager(s) (the system health manager). The challenge in defining this local health management is to ensure that the local diagnosis and mitigation are globally consistent.

System Health Manager (SHM) manages the overall health of the System (Component Assembly). The CLHM processes hosted inside each of the components report their input (alarms monitored events) and output (mitigation action) to the System Health Manager. It is important to know the local mitigation action because it could affect how the faults cascade through the system. Thereafter, the SHM is responsible for the identification of root failure source(s)[5]. Once the fault source is identified (diagnosed), an appropriate mitigation strategy could be employed. This as mentioned earlier is the topic of ongoing investigations.

4. Overview of ARINC Component Model (ACM)

The ARINC Component Model (ACM) [11],[12] is built upon the capabilities of the ARINC-653 [1] standard (see Appendix). ACM follows the MIC approach (see section 2) and borrows concepts from other software component models, notably from the CORBA Component Model (CCM) [25] with a focus on precisely defined component interaction semantics, enabling timing constraints and allowing component interactions to be monitored effectively.

Figure 2 illustrates the main features of ARINC Component Model. A component can have four different kinds of interaction ports - consumer port, publisher port, provided interface port (similar to a facet in CCM) and required interface port (similar to a CCM receptacle). A publisher port is a source of events: this port is used to produce events that will be consumed by another component/s. A publisher port needs to be triggered to publish an event (probably read from some internal state variable or a hardware source). This triggering can be either periodic or aperiodic (sporadic). While, a periodic publisher is triggered at regular intervals by a clock, an aperiodic publisher is invoked (sporadically) by an internal method of the component, possibly the implementation code belonging to another port.

A consumer port, as the name suggests, acts as a sink for events. Like a publisher port, it can be triggered periodically (by a clock) or aperiodically (by the arrival of an event) to consume an event. While an aperiodic consumer consumes

[5]We allow multiple failure mode hypotheses.

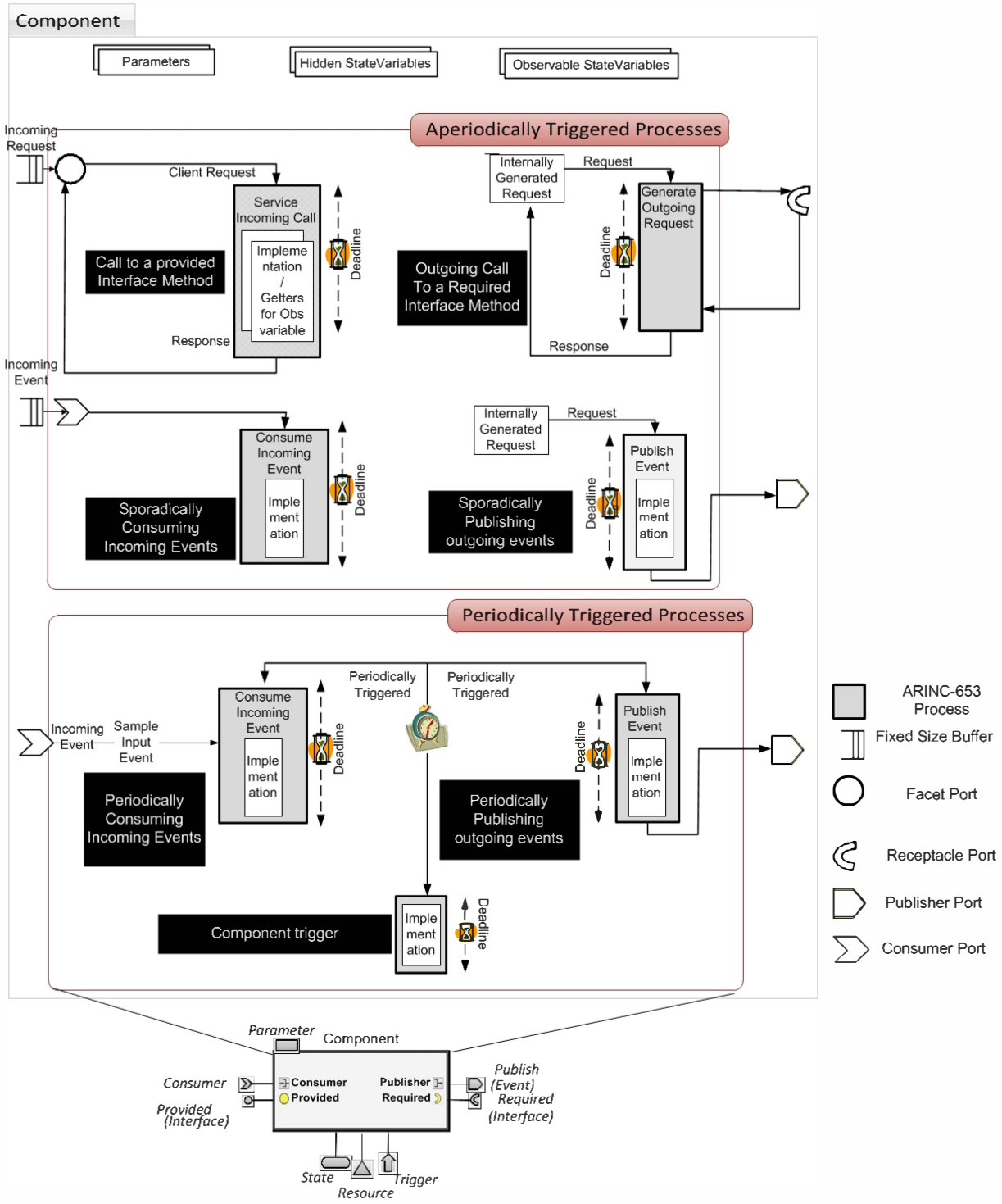


Figure 2. ARINC Component Model

all the events published by its publisher on a FIFO basis (destructive read), a periodic consumer samples the events published at a specified rate (nondestructive read).

A provided interface port or facet contains the implementation for the methods defined in the provided interface and services the request issued on these interfaces by a receptacle. The incoming client requests are queued by the middleware and are serviced by the provided port's implementation in FIFO order.

Two additional concepts exist in ACM as compared to the CCM: state variables, which are similar to attributes in CCM but cannot be modified from outside component, and component triggers, which are internal, periodically activated methods within a component that can be used for internal book-

keeping and checking state invariants.

The implementation methods associated with the component trigger and interaction ports (publisher, consumer, facet, and receptacle) are initialized as ARINC-653 processes. They have to finish their unit of work within a specified deadline. This deadline can be qualified as HARD (strict) or SOFT (relatively lenient). A HARD deadline violation is an error that requires intervention from the underlying middleware. A SOFT deadline violation results in a warning.

Like the deadline, the models can specify another property that the implementations must respect: *contracts*. These contracts are expressed as pre-conditions and/or post-conditions. Any contract violation results in an error. This concept is based upon the logic system developed by Hoare [16]. The key feature of this logic is the concept of assertions of the form $\{pre\}P\{post\}$ commonly known as *Hoare Triple*, where P is a computer program, pre is a pre-condition that is assumed to be true before the program is executed, and $post$ is the post-condition that is true after the program is executed.

Component Interactions

While each component and its associated ports, states, and internal triggers can be individually configured, an assembly is not complete until the interactions between the ports of all components have been configured. The association between the ports depends on their type (synchronous/asynchronous) and the event/interface type associated with the port. Two kinds of interactions: (1) asynchronous interactions and (2) synchronous interactions are possible between components. The possible combination of these interactions with periodic and aperiodic triggering of processes that are bound to the respective ports gives rise to a richer set of behaviors compared to CCM.

Asynchronous Interactions: These interactions occur when a publish port of a component is connected to a consumer port of another component. While a consumer can be connected to only one publisher, a publisher may be connected to one or more consumers. Strict type matching on the event type is required between the publisher and its consumers.

A periodic consumer always exhibits sampling behavior. Even if the rate of the publisher is indeterminate, for example if the publisher is aperiodic, setting the period of the consumer ensures that the events from the publisher are sampled at a specific rate. When the interacting publisher and consumer both are periodic, the value of the consumer's period relative to the publisher's determines if the consumer is over-sampling (higher rate of consumption or lower period compared to publisher) or under-sampling (lower rate of consumption or higher periodicity compared to publisher).

Interaction between a periodic publisher and an aperiodic consumer is indicative of a pattern where the sink or the consumer is reactive in nature. In such a case, the consumer port stores incoming published events in a queue, which are consumed in a FIFO manner. If the queue size is configured appropriately, this allows the consumer to operate on all of the events received.

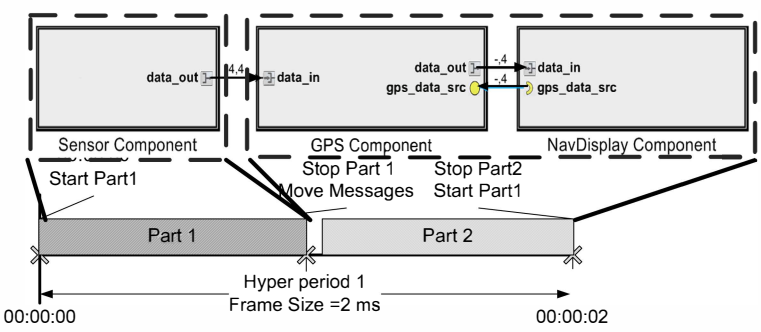
The case for interaction between an aperiodic publisher and an aperiodic consumer is similar to the one between a periodic publisher and an aperiodic consumer.

Synchronous Interactions: This interaction implies call-return semantics: the caller component 'calls out' via the required interface port to the connected provided interface port of the callee component. A required interface port can be associated with a provided interface port of an identical interface type. A provides port can be associated with one or more requires ports. Because of the synchronous nature of these interactions, the deadline of required interface method (i.e. the caller) must be greater than the deadline value for the provided interface method (i.e. the callee).

Synchronous ports in this model are always aperiodic. The interaction patterns observed in synchronous ports is borrowed from CCM. The key difference is deadline monitoring. The default type of interaction is call-return or two-way communication i.e. the required interface port waits for the provided interface port to finish its operation and return the results.

Modeling and Design Environment: The framework implementing ACM comes with a modeling language that allows the component developers to model a component and the set of services that it provides independent of actual deployment configuration, enabling preliminary constraint based verification of the system for well-formedness. An example for well-formedness is that each required port must be connected to precisely one provided port. Once fully specified, the component model captures the component's real-time properties and resource requirements. It also captures the internal data flow and control flow within the component. System integrators configure models of software assemblies specifying the architecture of the system built from interacting components.

While specifying component models in the modeling environment, developers can also specify local monitors and local health management actions for each component (described in sections 5 and 6). Once the assembly has been specified, system integrators are required to specify the models for system-level health management (described later in section 7). During the deployment and integration process, system integrators associate each component with an ARINC-653 partition. Thereafter, code generation tools help the integrators to generate non-functional glue code and find a suitable partition schedule and deploy the assembly. The developers write the functional code for each component using only the exposed interfaces provided by the framework. They are expected not to invoke the underlying low-level platform (APEX) services directly. Such restrictions enable us to use the well-defined semantics of specified interaction types between the compo-



Period(secs)	Component	Port	WCET secs
0.004	Sensor	data out	0.004
0.004	GPS	data in -after processing sends an event to Display	0.004
Sporadic	GPS	gps data src.GetGPSData	0.004
Sporadic	NavDisplay	data in -after processing calls GetGPSData	0.004
Sporadic	NavDisplay	gps data src.GetGPSData	0.004

```
CPU=1
HYPERPERIOD = 0.002
PARTITION_NAME = Partition2
PARTITION_NAME = Partition1
Partition2_EXECUTABLE =  ./Partition2/Partition2
Partition1_EXECUTABLE =  ./Partition1/Partition1
/////////SCHEDULING INFORMATION BEGINS/////////
Partition1_SCHEDULE = 0, 0.001
Partition2_SCHEDULE = 0.0012, 0.0008
/////////SCHEDULING INFORMATION ENDS/////////
Partition2_SAMPLINGPORT= Partition2_SP_0
Partition1_SAMPLINGPORT= Partition1_SP_0
////////////////////////////////////////////////////
Partition2_SP_0_MAXMESSAGESIZE = 36
Partition2_SP_0_REFRESHPERIOD = 0.004
Partition2_SP_0_DIRECTION = DESTINATION
////////////////////////////////////////////////////
Partition1_SP_0_MAXMESSAGESIZE = 36
Partition1_SP_0_REFRESHPERIOD = 0.004
Partition1_SP_0_DIRECTION = SOURCE
////////////////////////////////////////////////////
CHANNEL_NAME= Channel0
Channel0_SOURCE = Partition1_SP_0
Channel0_DESTINATION = Partition2_SP_0
```

Figure 3. GPS Software Assembly used in the case study - Unit of time is seconds.

nents and analyze the system failure propagation at design time before deployment. This in turn allows us to generate the necessary diagnosis procedures required. This is explained later in section 7. Thus during the deployment and integration process, code generators can also generate the required health management framework. The generated code can be later compiled and executed on the runtime system.

Example

Figure 4 shows an assembly of three components deployed on two ARINC partitions. We will use this example in the case study later on. Connections between two ports have been annotated with the (periodicity, deadline) pair, measured in milliseconds, of the downstream port. Partition 1 contains the Sensor Component. The sensor component publishes an event every 4 milliseconds.

Partition 2 contains the GPS and Navigation Display component. The GPS component consumes the event published by sensor at a periodic rate of 4 milliseconds. Afterwards it publishes an event, which is sporadically consumed by the Navigation Display (abbreviated as display). Thereafter, the display component updates its location by using getGPSData provided interface of the GPS Component. The publish-consume connection 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.

This figure also describes the periodic schedule followed by the partitions, overseen by a controller process called Module Manager [11]. This schedule is repeated every 2 ms (hyperperiod). In each cycle, Partition 1 runs with a phase of 0 ms for 1 ms (duration). Partition 2's phase is 1.2 ms. It runs for 0.8 ms (duration). This schedule ensures that the two partitions

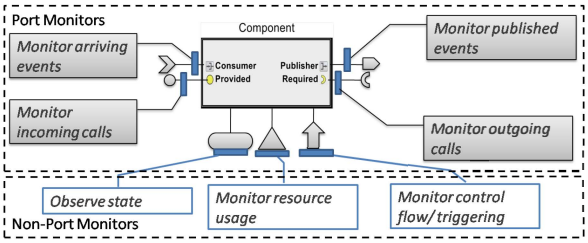


Figure 4. Component Monitoring

are temporally isolated.

5. Discrepancy Detection/Monitoring Specifications

The health of the software system/assembly and its individual components can be tracked by deploying multiple monitors throughout the system. Each monitor checks for violations of a property or constraint that is local to a port or a component. The status of these monitors is reported to Health Managers at one or more levels (Component or System) to take the appropriate mitigation action. The modeling language allows system integrators to define these monitors and declare whether they should be reported at the local or the system level. Figure 4 summarizes different places (or ports) where a component's behavior can be monitored to detect discrepancies. Based on these monitors, following discrepancies can be currently identified:

- *Lock Time Out*: 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 timeout it results in starvation. 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 de-

<**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 1. Monitoring Specification. Comments are shown in italics.

Issued By	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
Following actions can be issued only by a System health manager.		
SHM	RESET	Stop all operations, initialize state of component, clear all queues, start all periodic operations
SHM	CHECKPOINT	Save component state
SHM	RESTORE	Restore component state to the last saved state

Table 2. Component and System Health Manager Actions. Note that STOP for all process of a component in combination with start of processes from a redundant component can be used to reconfigure the system. The network link from the redundant component should be created at system initialization time.

signer.

• *Data Validity violation* (only applicable to consumers): Any 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 in the user code can be abstracted by the software developer as an error condition which they can choose to report to the framework. Any unreported error is recognized as a potential unobservable discrepancy.

• *Post-condition Violation*: Similar to preconditions, but these conditions are checked after the execution of function associated with the component port.

• *Deadline Violation*: Any execution started 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), (2) via a simple expression language (e.g. conditions). 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 value). Table 1 presents a summary.

6. COMPONENT-LEVEL HEALTH MANAGEMENT

A Component Level Health Manager (CLHM), as the name suggests, observes the health of a component. The operation of a CLHM can be specified as a state machine in the modeling environment. It can be configured to react with a mitigation action from a pre-defined set in response to violations observed by component monitors. Formally, a health manager can be described as a timed state machine $HM =< S, s_i, M, Z_{\tau^+}, T, A >$, where

- S is the set of all possible states for the health manager.
- $s_i \in S$ is the singleton initial state.
- M is the set of all monitored events that are reported to the health manager by a component process or the framework.
- Z_{τ^+} is the set of all events generated due to passage of time.
- A is the set of all possible mitigation actions issued by the health manager. Currently, supported mitigation actions are specified in Table 2.
- $T : S \times (M \cup Z_{\tau^+}) \rightarrow A \times S$ is the set of all possible transitions that can change the state of the manager due to passage of time or the arrival of an input event. To ensure a non-blocking state machine, the framework assumes a default

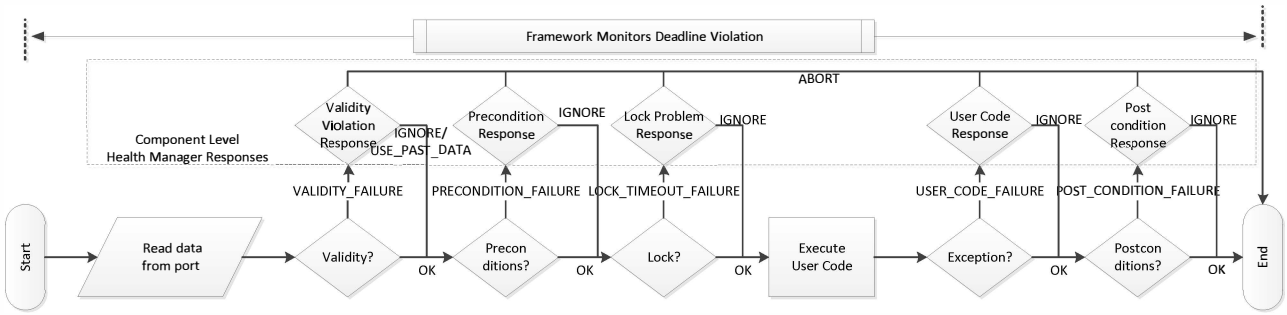


Figure 5. Flow chart describing sequence of monitors and health manager response for a consumer port.

self-transition with the IGNORE action if the health manager receives an event which it cannot process in the current state.

The process associated with the health manager is sporadically triggered by events generated either by the framework (for resource and deadline violation) or by the port monitors associated with the process. Each monitor checks if its specified condition is being satisfied. Upon detecting a violation, the monitors report to the component-level health manager. The CLHM's internal state machine tracks the component's state and issues mitigation actions. Processes that trigger the health manager can block using a blackboard to receive the health manager action[6]; they are finally released when the health manager publishes a response (mitigation action) on their respective blackboard.

Example: Execution Sequence of Generated Monitors and Component Health Manager Figure 5 shows the flowchart of the code generated to handle incoming messages on a consumer port. The shaded gray decision boxes are associated with the generated monitors. The failed monitored discrepancy is always reported to the local component health manager. Deadline violation is always monitored in parallel by the runtime framework. The white boxes are the possible actions taken by the local health manager.

7. System-Level Health Management

In our implementation, the System Health Manager (SHM) is a collection of three different components, shown in figure 6. These components can either be deployed in a separately reserved system module, or they can be deployed in a module shared by other components in the system assembly. The aggregator component is responsible for receiving all the alarm inputs, including the local component health manager decisions and passing them to the diagnosis engine. The aperiodic consumer inside the diagnosis engine runs in an aperiodic ARINC-653 process, which is triggered by the alarms sent by the aggregator. The third component is the response engine - this component is still under development.

The diagnosis engine uses a timed fault propagation (TFPG) model. A TFPG is a labeled directed graph where nodes represent either failure modes, which are fault causes, or discrepancies, which are off-nominal conditions that are the effects of failure modes. Edges between nodes in the graph capture the effect of failure propagation over time in the underlying dynamic system. To represent failure propagation in multi-mode (switching) systems, edges in the graph model can be activated or deactivated depending on a set of possible operation modes of the system. Appendix B provides a brief overview of TFPG.

The diagnosis engine uses the TFPG model of the software assembly under management to reason about the input alarms and the local responses received from different component level health manager. It then hypothesizes the possible faults that could have generated those alarms. As more information becomes available, the SHM (using the diagnosis engine) improves its fault-hypothesis as needed, which can then potentially be used to drive the mitigation strategy at the system level. Currently, available System level mitigation actions are listed in Table 2. However, this list is not final as system-level mitigation approaches are subject of ongoing investigations.

Creating the System Level Fault Propagation Model for System-Level Diagnosis

The fault propagation model for the entire system involves capturing the propagations within each component as well as capturing the propagations across component boundaries. While the latter can be automatically derived from the interactions captured by the software assembly (via component ports) the former can be derived from the interactions captured by the data/control flow model inside each component.

This automatic derivation of fault propagation from component and assembly models is possible because the end-points of these interactions - the component ports - exhibit a well defined behavior/interaction pattern[7]. This pattern is dependent on the specific port-type - Publisher, Consumer, Provides Interface, Requires Interface - and is somewhat inde-

[6]Blackboards are primitive, shared-memory type inter-process communication structures implemented by ARINC-653.

[7]Formal description of these interaction semantics is available in the appendix of the related technical report [13]

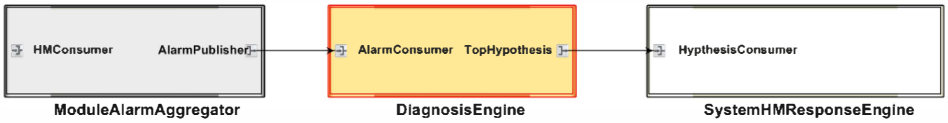


Figure 6. Components belonging to the system health manager.

pendent of the additional properties - data/event types, interfaces/methods, periodicity, deadline - that customize a port. Hence, if a template fault propagation model can be constructed for each of the different port-types, then using the interactions captured in the control/data flow model of the component and the assembly model of the system, the fault propagation graph for the entire system can be generated.

In principle, this approach is similar to the failure propagation and transformation calculus described by Wallace [28], which showed how architectural wiring of components and failure behavior of individual components can be used to compute failure properties for the entire system.

The template fault propagation model for each kind of interaction port deals with:

- Failures Modes that represent the failures originating from within the interaction port
- Monitored Discrepancies whose presence is detected through the Health Monitor Alarms
- The Unmonitored / silent Discrepancies whose presence is not detected through alarms
- The Input Discrepancy ports that represent entry points of failure effects from outside the interaction port
- The Output Discrepancy ports that represent exit points of failure effects to outside the interaction port
- The failure propagation links between the entities described above
- The Mode Variables that enable/disable a failure propagation edge based on their value which is set by the Component Health Manager's Response

The Failure Modes / Discrepancies are directly related to the list of monitors described in section 5. These include failure modes / discrepancies related to one or more of the following violations, failures, and problems - LOCK_Problem, Validity violation, Pre-condition failure, User code failure, Post-Condition failure, Deadline violation. The Mode Variables are related to the Component Health Manager's response to the errors detected by monitors - LOCK_Problem_Response, Validity_Violation_Response etc. The Input/Output Discrepancy ports list includes various manifestations of the problems listed above - No/ Late/ Invalid Data Published, No/ Late/ Invalid Return Data, Bad Input/Output Data, No Invoke, No Update etc.

Figure 7 captures the failure propagation template model of a periodic publisher and a periodic consumer. Additionally, it captures the failure interaction (red lines) between the publisher and consumer. In any component, the exact number and type of the Failure Modes, Monitored/Unmonitored discrepancies, Input/Output ports and the failure propagation links between them is determined by specific type of the interaction port - Publisher / Consumer / Provides Interface / Requires Interface. It should also be noted that sometimes it might not be possible to monitor some of the failures / alarms mentioned above. In such cases, these observed discrepancies are turned into unobserved discrepancies and the fault effect propagates through the discrepancy without raising any observation (alarm). The resulting template failure propagation model captures: (1) The effect of failures originating from other interaction-ports, (2) The cascading effects of failures within the interaction port, and (3) The effect of failures propagating to other interaction-ports.

As discussed earlier in this section, the component failure propagation model is generated by an algorithm, automatically, by instantiating the appropriate TFPG template-model for each interaction-port in the component. Thereafter, the information in the component's data/control flow model is used to generate the failure propagation links between the TFPG models of the interaction-ports within the same component. These failure propagation links connect input and output discrepancy ports in these TFPG models. Finally, the system level failure propagation model is generated by using the interaction information in the assembly model. Each link in the assembly model is translated into one or more failure propagation links between the TFPG models of the appropriate interaction-ports belonging to different components.

Example: Figure 7 shows a small portion of the failure propagation model between two components for the example described in section 4, figure 4. It shows the failure interactions (red lines) between a publisher and consumer. While the detailed failure propagation template-model of the publisher and consumer port are encapsulated within the box, the output and input discrepancy ports of the two models are connected through failure propagation links that cut across the boxes. A high level view of the full TFPG model for this example is shown in Figure 11.

As discussed in section 4, asynchronous interaction between a publisher port and a consumer port produces a fault propagation in the direction of data/event flow i.e. from the publisher to the consumer, while the synchronous (blocking) interaction pattern between a Requires interface and its corresponding Provider interface involves fault propagation in both directions. The fault propagation within a component is captured through the propagations across the bad updates on the state variables within the component, observed as pre-condition or

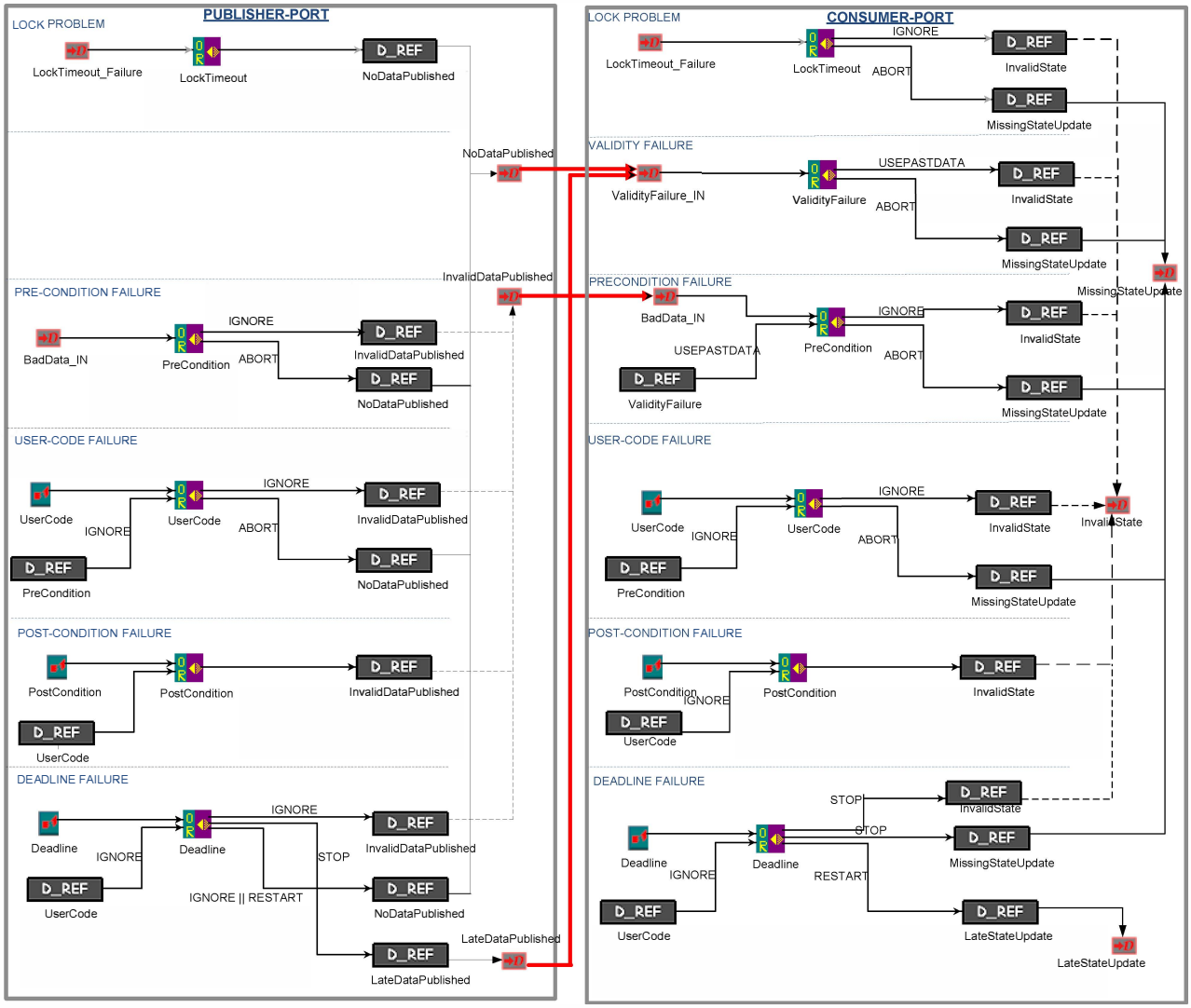


Figure 7. TFPG model of periodic publisher port and a periodic consumer port. Failure Propagation between the publisher and consumer is captured through bold red lines. In the template TFPG model of the publisher/consumer, the horizontal dotted lines separate one pattern from another. Root Nodes in each pattern are representative of either source of fault (Failure Mode) or cascading effect of another failure within the interaction port (Reference to a preceding discrepancy) or outside of the interaction port (Discrepancy Port).

post-condition monitors on the interfaces/interactions ports that update or read from those state variables.

8. Case Study

In this case study we consider the example of the GPS assembly discussed in section 4. First, we describe the nominal execution of the system. Then, we discuss component level health management and system-level diagnosis using two fault scenarios. This case study does not cover system level mitigation.

Baseline: No Fault. Figure 8 shows the timed sequence of events as they happen during the first frame of operation. These sequence charts were plotted using the plotter package from OMNeT++[8]. 0^{th} event marks the start of the module manager, which then creates the Linux processes for the two partitions. Each partition then creates its respective (APEX) processes and signals the module manager. This all happens before the frames are scheduled. After the occurrence of 0^{th} event, module manager signals partition 1 to start. Upon start, partition 1 starts the ORB process that handles all CORBA-related functions. It then starts the sensor health manager. Note that all processes are started in an order based on priority. Finally, it starts the periodic sensor process at event number 8. The sensor process publishes an event at event number

[8] http://www.omnetpp.org/

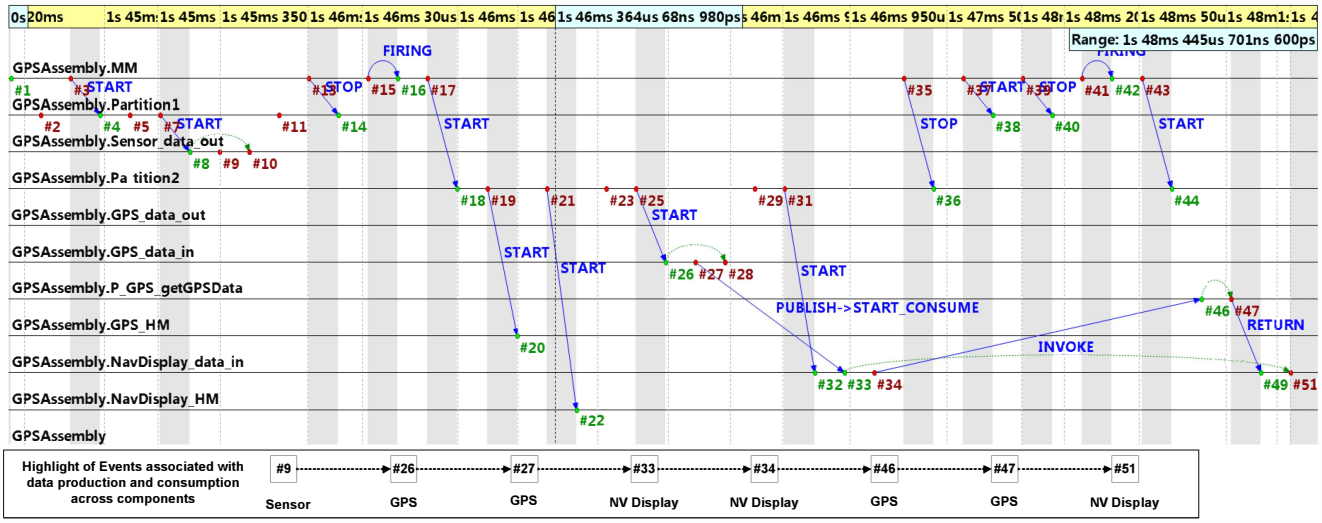


Figure 8. Sequence of Events for a no-fault case. The scale is non-linear.

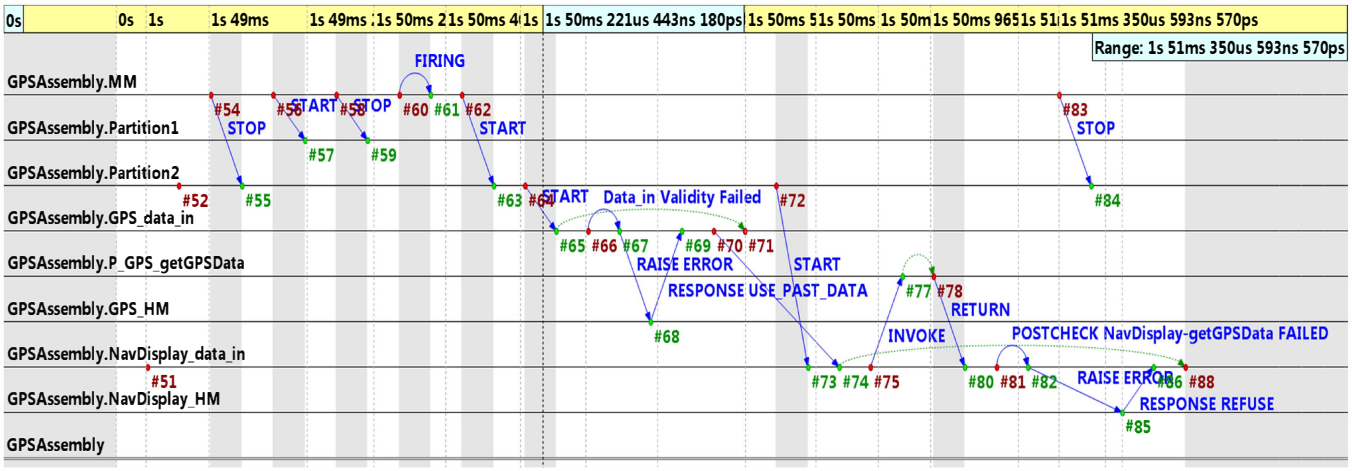


Figure 9. Sequence of events showing a fault scenario where GPS state is corrupted because the sensor does not update its output as expected. The sensor component is missing from the time line because it does not produce any event. The sequence of events also shows local health management action in the GPS and Nav Display.

9 and finishes its execution at event number 10. After 1 millisecond since its start, partition 1 is stopped by the module manager at event number 14. Immediately afterwards, partition 2 is started. Partition 2 starts all its CORBA ORB process and health managers at the beginning of its period. At event 26, partition 2 starts the periodic GPS consumer process. It consumes the sensor event at event 27. At event 27, GPS publisher process produces an event and finishes its execution cycle at 28. The production of GPS event causes the sporadic release of aperiodic consumer process in Navigation Display (event 33). The navigation process uses remote procedure call to invoke the GPS get data ARINC process. The GPS data value is returned to navigation process at event 49. It finishes the execution at event 51. Partition 2 is stopped after 1 millisecond from its start. This marks the end of one frame. Note that these events do not capture the internal functional logic of the GPS algorithm. Moreover, the claim of No-fault in this sequence of events is made because of the absence of any violation of component health monitors.

Fault Scenario: For the next two subsections we consider a scenario in which Sensor (figure 4) stops publishing data. First we describe the local component level health management action, which includes local detection as well as mitigation. Then we will show an example of system level diagnosis. System level mitigation has not been included in this example, as it still work in progress.

Component Level Health Management Example

Validity Violation at GPS Consumer Port. Sensor publishes an event every 4 milliseconds in the nominal condition. In this experiment, we injected a fault in the code such the sensor misses all event publications after its first execution. Figure 9 shows the experiment events that elapsed after the

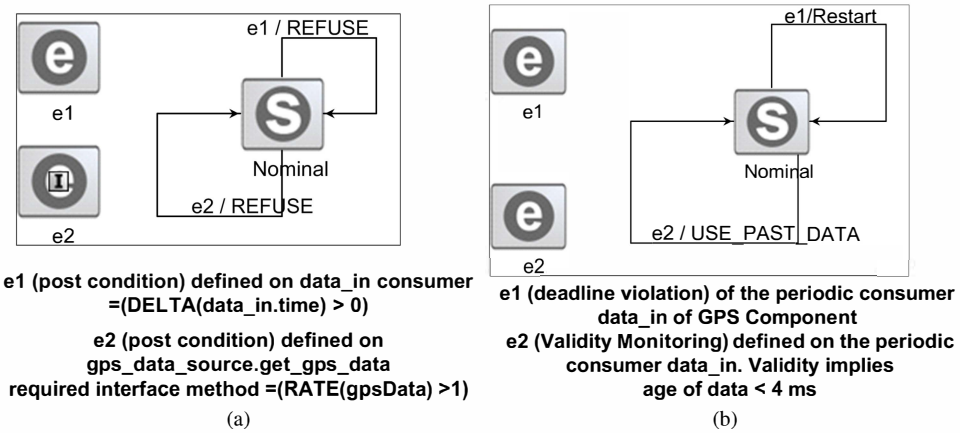


Figure 10. (a) CLHM for NavDisplay. It can be triggered by either event e1 or event e2; the programmed mitigation response is to refuse or abort the call. (b) CLHM for GPS. It can be triggered by either event e1 or event e2. Here Restart is a macro for STOP action followed by the START action.

sensor fault injection. As can be seen in the figure, there is no activity in Partition1 because of the sensor-fault (event # 57 to 59). The GPS process is started by partition 2 at event 65. At this time (event #66), the validity condition specified in the method that handles the incoming event fails. This condition checks the Boolean value of a validity flag that is set by the framework every time the sampling port is read. This validity flag is set to false if the age of the event stored in the sampling port is older than the refresh period specified for the sampling port (4 milliseconds in this case). Upon detection, the GPS process raises an error at event #67, which causes the release of GPS health manager at event #68. In this case, the GPS health manager (see figure 10) publishes a USE_PAST_DATA response back at event #68. The USE_PAST_DATA response (received in the data_in process at event #69) means that the process can continue and use the previously cached value.

Bad GPS Data at NavDisplay Port The fault introduced due to the missing sensor event and the GPS's response of use past data (event #69) results in a fault in the Navigation-Display component. Event numbers 73 to 88 in Figure 9 capture the snapshot corresponding to this experiment. The GPS's getGPSData process sends out bad data at event #78 when queried by the navigation display at event #75 using the remote procedure call. The bad data is defined by the rate of change of GPS data being less than a threshold. This fault simulates an error in the filtering algorithm in the GPS such that it loses track of the actual position because the sensor data did not get updated. . At event #81, the post condition check of the remote procedure call is violated. This violation is defined by a threshold on the RATE of change of current GPS data compared to past data (last sample). The navigation display component raises an error at event #82 to its CLHM. At event #86, it receives a REFUSE response from the health manager (see figure 10(a)). The REFUSE response means that the process that detected the fault should immediately abort further processing and return cleanly. The effect of this action is that the navigation's GPS coordinates are not updated as the remote procedure call did not finish without error. The next subsection discusses the system level health management actions related to this fault cascade scenario.

System Level Health Management Example

Figure 11 shows the high-level TFPG model for the system/assembly described in figure 4. The detailed TFPG-model specific to each interaction pattern is contained inside the respective TFPG component model (brown box). The figure shows failure propagation between the Sensor publisher (Sensor_data_out) and GPS consumer(GPS_data_in), the GPS publisher (GPS_data_out) and NavDisplay consumer (NavDisplay_data_in), the requires method in NavDisplay(NavDisplay_gps_data_src_getGPSData) and the provides method in GPS (GPS_gps_data_src_getGPSData), the effect of the bad updates on state variables and the entities updating or reading the state-variables.

System Level Diagnosis Process: Figure 12 shows the assembly in figure 4 augmented with Component and System level Health Managers and the interaction between them. 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 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. The higher the metrics the more reasonable it is to expect the hypothesis to be the real cause of the problem. As more alarms are produced, the hypothesis are further refined. If the new alarms are supportive of existing hypotheses, they are updated to reflect the refinement in their metrics and alarm list. If the new

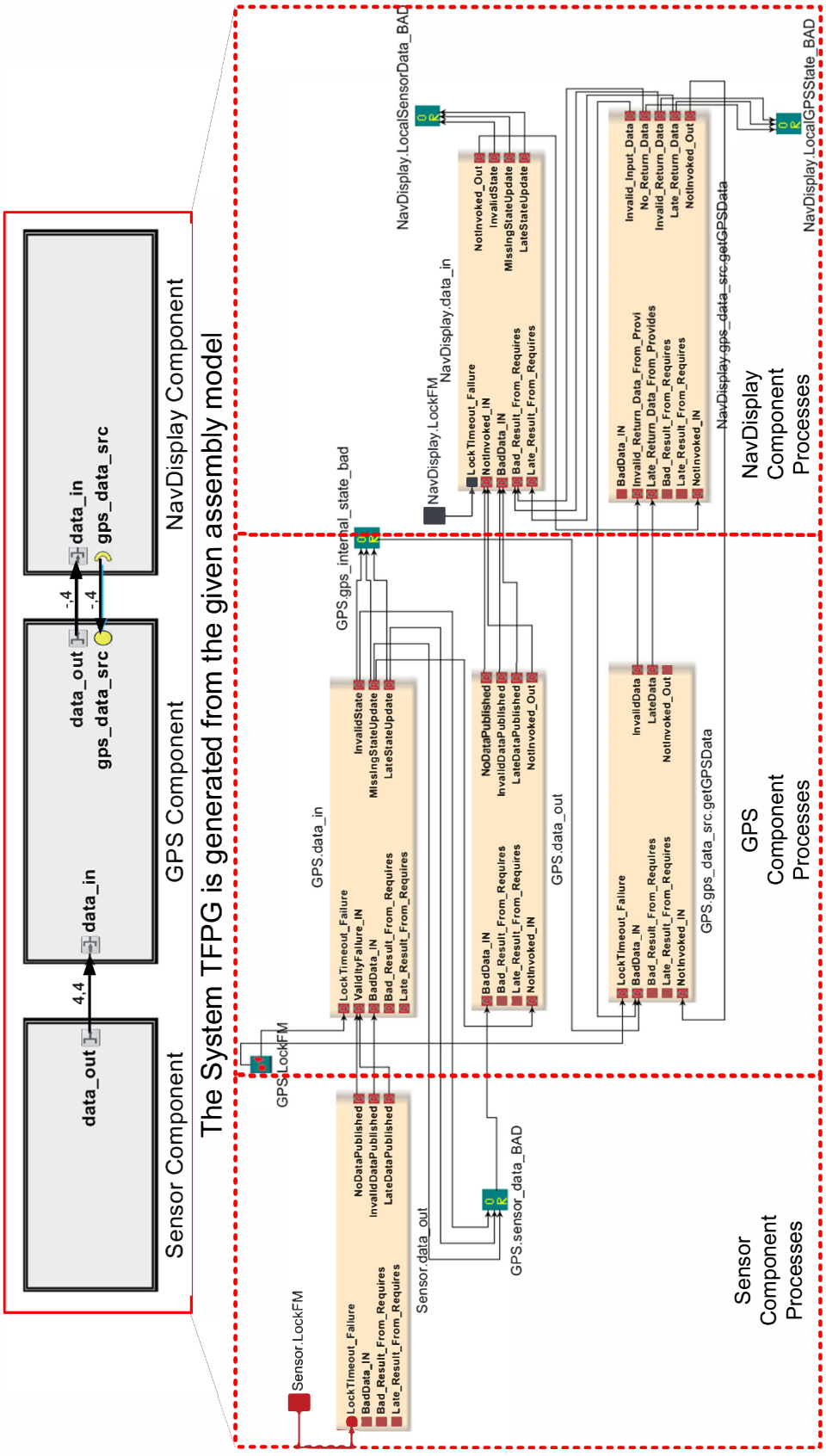
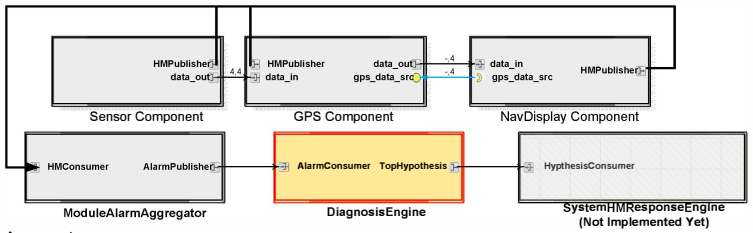


Figure 11. TFPG model for the assembly



Output Of Alarm Aggregator

```
28916:Partition3|1273281809.360706622|HME|RECEIVED Monitor: Error Code 2, Component 2, Process 7, Partition 1, Local HM Action 5, time 1273281808760746705
28916:Partition3|1273281809.360952393|HME|RECEIVED Monitor: Error Code 5, Component 3, Process 11, Partition 1, Local HM Action 0, time 1273281808761494007
28916:Partition3|1273281813.360637128|HME|RECEIVED Monitor: Error Code 2, Component 2, Process 7, Partition 1, Local HM Action 5, time 1273281812760731758
28916:Partition3|1273281813.360889186|HME|RECEIVED Monitor: Error Code 5, Component 3, Process 11, Partition 1, Local HM Action 0, time 1273281812761455453
28916:Partition3|1273281821.360642647|HME|RECEIVED Monitor: Error Code 5, Component 3, Process 11, Partition 1, Local HM Action 0, time 1273281820761304597
```

Output Of Diagnosis Engine

```
1.  ===============[ Alarm Monitor AM_GPS_data_in_VALIDITY_FAILURE Triggered at TIME = 24.3411 ============================
2.  ===============[ TFPG REASONSER INVOKED. TIME = 24.3411 ============================
3.  ================[ UPDATING ALARMS TRIGGERED.]================
4.  ====================[ DISCREPANCY ALARM DISC_GPS_data_in_VALIDITY_FAILURE [ AM_GPS_data_in_VALIDITY_FAILURE  TRIGGERED ]============
5.  ================[ Hypothesis Group 1 ]================
6.  Fault: FM_Sensor_data_out_USER_CODE      Component: GPSAssembly    failure rate: 0.000000 earliest time: 0.000000  latest time: 24.341104
7.  ------- Supporting Alarms  :DISC_GPS_data_in_VALIDITY_FAILURE [ AM_GPS_data_in_VALIDITY_FAILURE ]
8.  ------- Expected Alarms  :DISC_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE [ AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE ]
9.  ------- Plausibility: 100.000000        Robustness: 50.000000  FRMetric: 0
10. ================[ Hypothesis Group 2 ]================
11. Fault: Sensor__LOCK_PROBLEM      Component: GPSAssembly    failure rate: 0.000000 earliest time: 0.000000  latest time: 24.341104
12. ------- Supporting Alarms  :DISC_GPS_data_in_VALIDITY_FAILURE [ AM_GPS_data_in_VALIDITY_FAILURE ]
13. ------- Expected Alarms  :DISC_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE [ AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE ]
14. ------- Plausibility: 100.000000        Robustness: 50.000000  FRMetric: 0
15. ===============[ Alarm Monitor AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE Triggered at TIME = 24.3417 ====================
16. ===============[ TFPG REASONSER INVOKED. TIME = 24.3417 ============================
17. ================[ UPDATING ALARMS TRIGGERED.]================
18. ====================[ DISCREPANCY ALARM DISC_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE  [
AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE  TRIGGERED ]====================
19. ================[ Hypothesis Group 1 ]================
20. Fault: FM_Sensor_data_out_USER_CODE      Component: GPSAssembly    failure rate: 0.000000 earliest time: 0.000000  latest time: 24.341104
21. ------- Supporting Alarms  :DISC_GPS_data_in_VALIDITY_FAILURE [ AM_GPS_data_in_VALIDITY_FAILURE ]DISC_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE [
AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE ]
22. ------- Plausibility: 100.000000        Robustness: 100.000000        FRMetric: 0
23. ================[ Hypothesis Group 2 ]================
24. Fault: Sensor__LOCK_PROBLEM      Component: GPSAssembly    failure rate: 0.000000 earliest time: 0.000000  latest time: 24.341104
25. ------- Supporting Alarms  :DISC_GPS_data_in_VALIDITY_FAILURE [ AM_GPS_data_in_VALIDITY_FAILURE ]DISC_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE [
AM_NavDisplay_gps_data_source_getGPSData_POSTCONDITION_FAILURE ]
26. ------- Plausibility: 100.000000        Robustness: 100.000000        FRMetric: 0
```

Figure 12. This figures shows augmentation of the assembly shown in figure 4 with an alarm aggregator component, the diagnosis engine, and the system level response engine. Details of this last component are not in this paper, as it is the subject of our ongoing research. Also shown are the results from the alarm aggregator and the diagnosis engine.

alarms are not supportive of any of the existing hypotheses with the highest plausibility, then the reasoner refines these hypotheses such that hypotheses can explain these alarms.

Figure 12 also shows the TFPG-results for fault scenario under study. The initial alarm is generated because of data-validity violations in the consumer of the GPS component. When this alarm was reported to the local Component Health manager, it issued a response directing the GPS component to use past data (USE_PAST_DATA). While the issue was resolved local to the GPS component, the combined effect of the failure and mitigation action propagated to the Navigation Display component. In the Navigation Display component, a monitor observing the post-condition violation on a Required interface was triggered because the GPS-data validated its constraints. These two alarms were sent to the System Health Manager and processed by the TFPG-Diagnoser.

As can be seen from the results, the system correctly generated two hypotheses (figure 12, lines 20 and 24). The first hypothesis blamed the sensor component lock to be the root problem. The second hypothesis blamed the user level code in the sensor publisher process to be the root failure mode. In this situation the second hypothesis was the true cause. However, because in this example lock time out monitors were not specified the diagnoser was not able to reasonably disambiguate between the two possibilities.

9. Related Work

One notable approach to system health management for physical systems is to design a controller that inherently drives the system back in safe region upon failure of a system. This is the basis of goal-based control paradigm [29] that supports a deductive controller that is responsible for observing the plant's state (mode estimation) and issuing commands to move the plant through a sequence of states that achieves the specified goal. This approach inherently provides for fault recovery (to the extent feasible) by using the control program to set an appropriate configuration goal that attempts to negate

the problems caused by faults in the physical system. However, these control algorithms are themselves typically implemented in software and are therefore reliant on the fault-free behavior of related software components.

Formal argument for checking correctness of execution of a computer program based on a first order logic system was first presented by Hoare in [16]. Later this concept was extended to distributed systems by Meyer in [21], [17]. A contract implemented by Meyer specified the requires and ensure clauses as assertions specified by a list of boolean expressions. These assertions were specified as logic operations upon the value domain of the program variables and were compiled out in the running system. In ACM, these correctness conditions are specified by preconditions and post conditions, which can be defined over both the value-domain and temporal domain of program variables as well as the state variables belonging to the component. We envision that these checks are performed in real-time on the system. This is especially necessary because there is a high likelihood for software defects being present in complex systems that arise only under exceptional circumstances. These circumstances may include faults in the hardware system (including both the computing and non-computing hardware) - software is very often not prepared for hardware faults [13].

Conmy et al. presented a framework for certifying Integrated Modular Avionics applications build on ARINC-653 platforms in [9]. Their main approach was the use of 'safety contracts' to validate the system at design time. They defined the relationship between two or more components within a safety critical system. However, they did not present any details on the nature of these contracts and how they can be specified. We believe that a similar approach can be taken to formulate acceptance criteria, in terms of "correct" value-domain and temporal-domain properties that will let us detect any deviation in a component's behavior.

Nicholson presented the concept of reconfiguration in integrated modular systems running on operating systems that provide robust spatial and temporal partitioning in [22]. He identified that health monitoring is critical for a safety-critical software system and that in the future it will be necessary to trade-off redundancy based fault tolerance for the ability of "reconfiguration on failure" while still operational. He described that a possibility for achieving this goal is to use a set of lookup tables, similar to the health monitoring tables used in ARINC-653 system specification, that maps trigger event to a set of system blue-prints providing the mapping functions. Furthermore, he identified that this kind of reconfiguration is more amenable to failures that happen gradually, indicated by parameter deviations.

Goldberg and Horvath have discussed discrepancy monitoring in the context of ARINC-653 health-management architecture in [14]. They describe extensions to the application executive component, software instrumentation and a temporal logic run-time framework. Their method primarily depends on modeling the expected timed behavior of a process, a partition, or a core module - the different levels of fault-protection layers. All behavior models contain "faulty states" which represent the violation of an expected property. They associate mitigation functions using callbacks with each fault.

Sammapun et al. describe a run-time verification approach for properties written in a timed variant of LTL called MEDL in [26]. They described an architecture called RT-MaC for checking the properties of a target program during run-time. All properties are evaluated based on a sequence of observations made on a "target program". To make these observations all target programs are modified to include a "filter" that generates the interesting event and reports values to the event recognizer. The event recognizer is a module that forwards the events to a checker that can check the property. Timing properties are checked by using watchdog timers on the machines executing the target program. Main difference in this approach and the approach of Goldberg and Horvath outlined in previous paragraph is that RT-MaC supports an "until" operator that allows specification of a time bound where a given property must hold. Both of these efforts provided valuable input to our design of run-time component level health management.

10. SUMMARY

This paper presented our first steps towards building a Software Health Management technology that extends beyond classical software fault tolerance techniques. In the approach, we briefly discussed our framework first that combines component-oriented software construction with a real-time operating system with partitioning capability (ARINC 653). Based on this framework, we defined an approach for 'Component-level Software Health Management' and created a model-based toolsuite (modeling tool, generators, and software platform) that supports the model-driven engineering of component-based systems with health management services.

We also showed how we can perform system-level diagnosis, which is required for system level health management, where faults occur in and propagate across many components. Our diagnosis procedure is based on a Timed Failure Propagation model for the system, automatically synthesized from the software assembly models. Our current work is focusing on extending the component level mitigation procedure to the system-level, where more sophisticated mitigation logic is necessary. We also plan to extend this work to the entire, larger system: a cyber-physical system, like a large sub-system of an aerospace vehicle, that may have its own, non-software failure modes. The challenge in that level is to integrate health management across the entire hardware / software ensemble. Additionally, we hope to leverage our work with distributed TFPG reasoners [20] and explore a distributed health management approach that addresses issues related to single point of failures, scalability, and other issues.

APPENDIX

1. BACKGROUND ON ARINC-653

The ARINC-653 software specification describes the standard Application Executive (APEX) kernel and associated services that should be supported by safety-critical real-time operating system (RTOS) used in avionics. It has also been proposed as the standard operating system interface on space missions [10]. The APEX kernel in such systems is required to provide robust spatial and temporal partitioning. The purpose of such partitioning is to provide functional separation between applications for fault-containment. A partition in this environment is similar to an application process in regular operating systems, however, it is completely isolated, both spatially and temporally, from other partitions in the system and it also acts as a fault-containment unit. It also provides a reactive health monitoring service that supports recovery actions by using call-back functions, which are mapped to specific error conditions in configuration tables at the partition/module/system level.

Spatial partitioning [14] ensures exclusive use of a memory region for a partition by an ARINC process (unless otherwise mentioned, a 'process' is meant to be understood as an 'ARINC Process' throughout this paper). It is similar to a thread in regular operating systems. Each partition has predetermined areas of allocated memory and its processes are prohibited from accessing memory outside of the partition's defined memory area. The protection for memory is enforced by the use of memory management hardware. This guarantees that a faulty process in a partition cannot ruin the data structures of other processes in different partitions. For instance, space partitioning can be used to separate the low-criticality vehicle management components from safety-critical flight control components. Faults in the vehicle management components must not destroy or interfere with the flight control components, and this property could be ensured via the partitioning mechanism.

Temporal partitioning [14] refers to the strict time-slicing of partitions, guaranteeing access for the partitions to the processing resource(s) according to a fixed, periodic schedule. The operating system core (supported by hardware timer devices) is responsible for enforcing the partitioning and managing the individual partitions. The partitions are scheduled on a fixed-time basis, and the order and timing of partitions are defined at configuration time. This provides deterministic scheduling whereby the partitions are allowed to access the processor or other hardware resources for only a predetermined period of time. Temporal partitioning guarantees that a partition has exclusive access to the resources during its assigned time period. It also guarantees that when the predetermined period of execution time of a partition is over, the execution of the partition will be interrupted and the partition itself will be put into a dormant state. Then, the next partition in the schedule order will be granted the right to execution. Note that all shared hardware resources must be managed by the partitioning operating system in order to ensure that control of the resource is relinquished when the time-slice for the corresponding partition expires.

2. BACKGROUND ON TFPG

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. Formally, a TFPG is represented as a tuple (F, D, E, M, A), where:

- F is a nonempty set of failure nodes.
- D is a nonempty set of discrepancy nodes. Each discrepancy node is of AND or OR type.[9]. Further, if a discrepancy is observable then it is associated with an alarm.
- $E \subseteq V \times V$ is a set of edges connecting the set of all nodes $V = F \cup D$. Each edge has a minimum and a maximum time interval within which the failure effect will propagate from the source to the destination node. Further, an edge can be active or inactive based on the state of its associated system modes.
- M is a nonempty set of system modes.
- A is a nonempty set of alarms.

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[15], [7],[6]. The model is used to derive efficient reasoning algorithms that implement fault diagnostics: fault source identification by tracing observed discrepancies back to their originating failure modes. The TFPG approach has been applied and evaluated for various aerospace and industrial systems[24]. More recently, a distributed approach has been developed for reasoning with TFPG[20].

ACKNOWLEDGMENTS

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.

REFERENCES

[1] "Arinc specification 653-2: Avionics application software standard interface part 1 - required services," Tech. Rep.

[2] "Mathworks, Inc., www.mathworks.com."

[9] An OR(AND) type discrepancy node will be activated when the failure propagates to the node from any (all) of its predecessor nodes.

[3] "Model-Driven Architecture," www.omg.org/mda.

[4] "Model-Integrated Computing," http://www.isis.vanderbilt.edu/research/MIC.

[5] "National Instruments," www.ni.com.

[6] 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*, vol. 58, no. 2, pp. 240–247, February 2009.

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

[8] R. Butler, "A primer on architectural level fault tolerance," NASA Scientific and Technical Information (STI) Program Office, Report No. NASA/TM-2008-215108, Tech. Rep., 2008. [Online]. Available: http://shemesh.larc.nasa.gov/fm/papers/Butler-TM-2008-215108-Primer-FT.pdf

[9] P. Conmy, J. McDermid, and M. Nicholson, "Safety analysis and certification of open distributed systems," in *International System Safety Conference,*, Denver, 2002.

[10] N. Diniz and J. Rufino, "ARINC 653 in space," in *Data Systems in Aerospace*. European Space Agency, May 2005.

[11] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan, "A real-time component framework: Experience with ccm and arinc-653," *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, pp. 143–150, 2010.

[12] A. Dubey, G. Karsai, and N. Mahadevan, "A component model for hard-real time systems: Ccm with arinc-653," *Softw., Pract. Exper.*, to Appear.

[13] ——, "Towards model-based software health management for real-time systems." Institute for Software Integrated Systems, Vanderbilt University, Tech. Rep. ISIS-10-106, August 2010. [Online]. Available: http://isis.vanderbilt.edu/node/4196

[14] A. Goldberg and G. Horvath, "Software fault protection with ARINC 653," in *Proc. IEEE Aerospace Conference*, March 2007, pp. 1–11.

[15] S. Hayden, N. Oza, R. Mah, R. Mackey, S. Narasimhan, G. Karsai, S. Poll, S. Deb, and M. Shirley, "Diagnostic technology evaluation report for on-board crew launch vehicle," NASA, Tech. Rep., 2006.

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[17] J.-M. Jézéquel and B. Meyer, "Design by contract: The lessons of ariane," *Computer*, vol. 30, no. 1, pp. 129–130, 1997.

[18] S. Johnson, Ed., *System Health Management: With Aerospace Applications.* John Wiley & Sons, Inc, Based on papers from First International Forum on Integrated System Health Engineering and Management in Aerospace, 2005. To Appear in 2011.

[19] M. R. Lyu, *Software Fault Tolerance.* John Wiley & Sons, Inc, 1995, vol. New York, NY, USA. [Online]. Available: http://www.cse.cuhk.edu.hk/~lyu/book/sft/

[20] N. Mahadevan, S. Abdelwahed, A. Dubey, and G. Karsai, "Distributed diagnosis of complex causal systems using timed failure propagation graph models," in *IEEE Systems Readiness Technology Conference, AUTOTESTCON*, 2010.

[21] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[22] M. Nicholson, "Health monitoring for reconfigurable integrated control systems," *Constituents of Modern System safety Thinking. Proceedings of the Thirteenth Safety-critical Systems Symposium.*, vol. 5, pp. 149–162, 2007.

[23] S. Ofsthun, "Integrated vehicle health management for aerospace platforms," *Instrumentation Measurement Magazine, IEEE*, vol. 5, no. 3, pp. 21 – 24, Sep. 2002.

[24] S. C. Ofsthun and S. Abdelwahed, "Practical applications of timed failure propagation graphs for vehicle diagnosis," in *Proc. IEEE Autotestcon*, 17–20 Sept. 2007, pp. 250–259.

[25] A. Puder, "MICO: An open source CORBA implementation," *IEEE Softw.*, vol. 21, no. 4, pp. 17–19, 2004.

[26] U. Sammapun, I. Lee, and O. Sokolsky, "RT-MaC: runtime monitoring and checking of quantitative and probabilistic properties," in *Proc. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 17–19 Aug. 2005, pp. 147–153.

[27] W. Torres-Pomales, "Software fault tolerance: A tutorial," NASA, Tech. Rep., 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8307

[28] M. Wallace, "Modular architectural representation and analysis of fault propagation and transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 141, no. 3, pp. 53–71, 2005.

[29] B. C. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, "Model-based programming of fault-aware systems," *AI Magazine*, vol. 24, no. 4, pp. 61–75, 2004.

BIOGRAPHY

Abhishek Dubey is a Research Scientist at the Institute for Software Integrated Systems at Vanderbilt University. He has nine years of experience in software engineering. He conducts research in theory and application of model-predictive control for managing performance of distributed computing systems, in design of fault-tolerant software frameworks for scientific computing, in practice of model-integrated computing, and in fault-adaptive control technology for software in hard real-time systems. He received his Bachelors from the Institute of Technology, Banaras Hindu University, India in 2001, and received his M.S and PhD from Vanderbilt University in 2005 and 2009 respectively. He has published over 20 research papers and is a member of IEEE.

Gabor Karsai is Professor of Electrical and Computer Engineering at Vanderbilt University and Senior Research Scientist at the Institute for Software-Integrated. He has over twenty years of experience in software engineering. He conducts research in the design and implementation of advanced software systems for real-time, intelligent control systems, and in programming tools for building visual programming environments, and in the theory and practice of model-integrated computing. He received his BSc and MSc from the Technical University of Budapest, in 1982 and 1984, respectively, and his PhD from Vanderbilt University in 1988, all in electrical and computer engineering. He has published over 100 papers, and he is the co-author of four patents.

Nagabhushan Mahadevan is a Senior Staff Engineer at the Institute for Software Integrated Systems (ISIS), Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, where his work is focused on using model-based techniques towards diagnosis, distributed diagnosis, software health management, adaptation of software-intensive systems and quality-of-service management. He received his M.S. degree in Computer Engineering and Chemical Engineering from the University of South Carolina, Columbia, and B.E.(Hons.) degree in Chemical Engineering from Birla Institute of Technology and Science, Pilani, India.