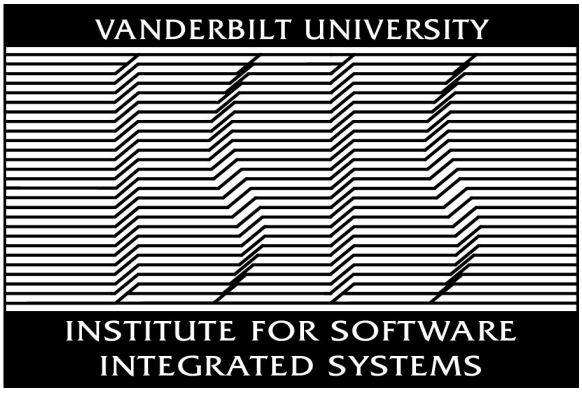


Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37203

Towards Model-based Software Health Management for Real-Time Systems

Abhishek Dubey Gabor Karsai Nagabhushan Mahadevan

TECHNICAL REPORT

ISIS-10-106

August, 2010

Towards Model-based Software Health Management for Real-Time Systems

Abhishek Dubey Gabor Karsai Nagabhushan Mahadevan

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

Abstract

The 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 can 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. The paper introduces the fundamental ideas for software health management, and then illustrates how these can be implemented in a model-based software development process, including a case study and related work.

1 Introduction

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

There is a well-established literature of software fault tolerance which carries of the techniques of hardware fault tolerance (like triple modular redundancy) to the software domain. We argue, however that software complexity has evolved to a new level where such techniques do not provide a sufficient technology anymore and new approaches are needed. While the architectural principles of software fault tolerance are clear, the complexity of anomaly detection, fault source isolation, and fault mitigation has grown to the point that by itself this has become a potential source of faults. In other words, the implementation of software fault tolerance may lead to faults.

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 component-based software and model-driven development.

We argue that software fault management can and should borrow 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 (off line), and fault prognostics (on-line or off-line). 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. The main differences between system health management and fault tolerant design has three aspects: (1) system health management deals with the entire system, not only with a single subsystem or component, (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) health management explicitly aims at predicting future faults from early precursor anomalies of those faults.

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 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 Backgrounds

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 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 behavior, 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, i.e. embedded in the running system.
4. 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 Basic 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 faults and failures.

To apply these techniques to software we must start by identifying the basic 'Fault Containment Units'. Arguably, one 'line of code' is not a very good abstraction for this purpose. Instead, we assume that *software systems are built from 'software components'*, where 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, as well as supports fault management.

There are various levels at which health management techniques can be applied: ranging from level of individual components, level of subsystems, to the whole system. Initially, we have focused on two levels of software health management: *Component level* (this is main focus of this paper) and the *System level*. Component-level health management (CLHM) for software components detects anomalies, identifies and isolates the fault causes of those anomalies (if feasible), prognosticate 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.

Implementing CLHM requires various monitoring functions that observe what is happening on the component's interfaces, including monitoring preconditions, postconditons, deadline violations; generating diagnosis results and, if possible, prognosis for the health of the component, and taking mitigating actions. We argue that such functions can be implemented by code that is generated automatically from some higher-level model of the CLHM. The next section provides some details about our approach. In a previous paper [8], we presented the design of a software component framework that can serve as a foundation for SHM in real-time systems. Subsequently, we have enhanced the design of the component framework to provide various monitoring capabilities. We have also implemented a system level diagnoser. A complete description of system level diagnoser is out of context of this paper.

4 Software Health Management Techniques

In course of our research, we recognized that in-spite of the apparent benefits of a component-based approach to development for safety-critical systems, little work has been done on applying these concepts to hard real-time systems [8]. Specifically, there appears to be a need for a component model and a corresponding software framework that introduces the principles and techniques of component-based development to ARINC-653 systems [1] with a focus on precisely defined component interaction semantics, enabling timing constraints and allowing component interactions to be monitored effectively. Next few paragraphs provide a brief summary of the developed component model.

4.1 The ARINC Component Model and Framework

The ARINC Component Framework (ACF) (described in detail in [8]) is built upon the capabilities of ARINC-653 [1], the state of the art operating system standard used in Integrated Modular Avionics. It borrows concepts from other software component frameworks, notably from the CORBA Component Model (CCM) [16].

Figure 1 illustrates the main features of ARINC Component Model. A component can have four different kinds of 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

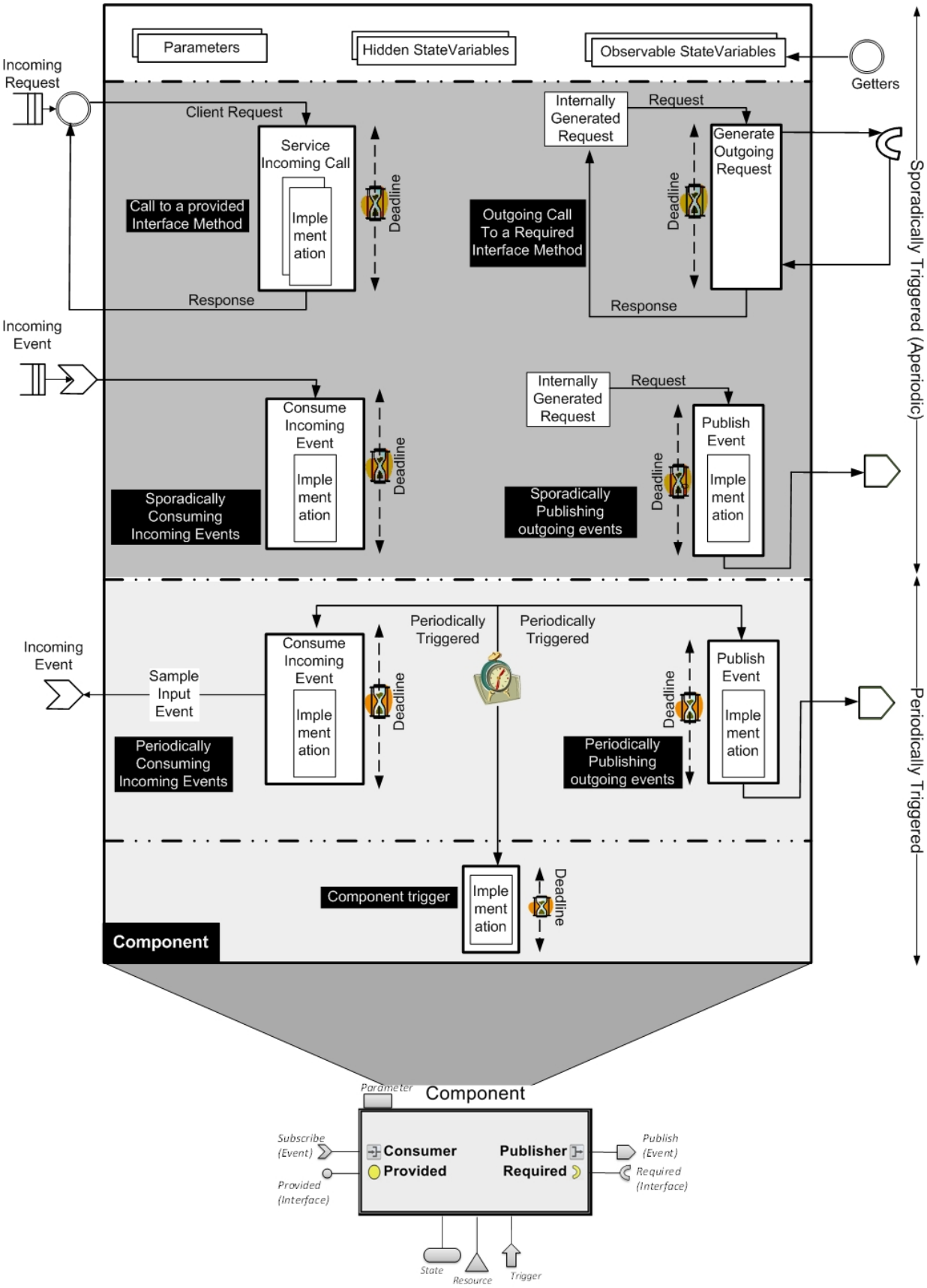


Figure 1: Component Model

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 to supply data , an aperiodic publisher is invoked (sporadically) by an internal method, possibly the implementation code of 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 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 the FIFO order. Two new concepts exist in our extension 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 bookkeeping and checking state invariants.

All the ports - publisher, consumer, facet, and receptacle - and the Component Trigger method 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, all implementations can specify another property that must be respected - *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 identified by Hoare [12]. The key feature of this logic are assertions of form $\{pre\}P\{post\}$ commonly known as *Hoare Triple*, where P is a computer program.

4.1.1 Component Interactions

While each component and its associated ports, states, internal triggers can be individually configured, an assembly is not complete until the interactions between the component ports is 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, asynchronous interactions and 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. 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 Requires port waits for the provides port to finish its operation and return the results.

Recently, we relaxed the restrictions on synchronous interactions to allow CORBA style one-way calls. When such methods are invoked, the Requires port performs a non-blocking call. It returns without waiting for the Provides port to finish its operation. There are no return values in such calls. However, one should note that even though the call is made in a non-blocking fashion it is different from an asynchronous interaction. While, a publisher does not fail if a consumer fails to consume the message properly, a one-way call via the middleware will result in an exception if the target provided port is not available.

Appendix A describes these interactions, their sequence and timing in detail.

4.2 Modeling and Design Environment

ACF 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 (see figure 2). This allows us to conduct preliminary constraint based analysis on the system. The model captures the component's real-time properties and resource requirements using a domain specific modeling language. System integrators then configure software assemblies specifying the architecture of the system built from interacting components.

Component developers can also specify local monitors and local health management actions for each component (described later in section 4.3). Once the assembly has been specified, system integrators are required to specify the models for system-level health management (described later in section 4.4). During the assembly process, code generation tools help the integrators to generate non-functional glue code 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 components and analyze the system failure propagation at design time before deployment. Thus during the deployment phase, code generators can also generate the required health management framework. The generated code can be later compiled and executed on the runtime system.

4.3 Anomaly 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 3 summarizes different places (or ports) where a component's behavior can be monitored to detect anomalies. These monitors check properties (contracts) associated with the following items:

- **Resource**: The framework implicitly monitors for any lock failure or deadline violation.
- **Data**: Any data token consumed by a consumer port has an associated expiry 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 ports. Moreover, developers can specify conditions 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

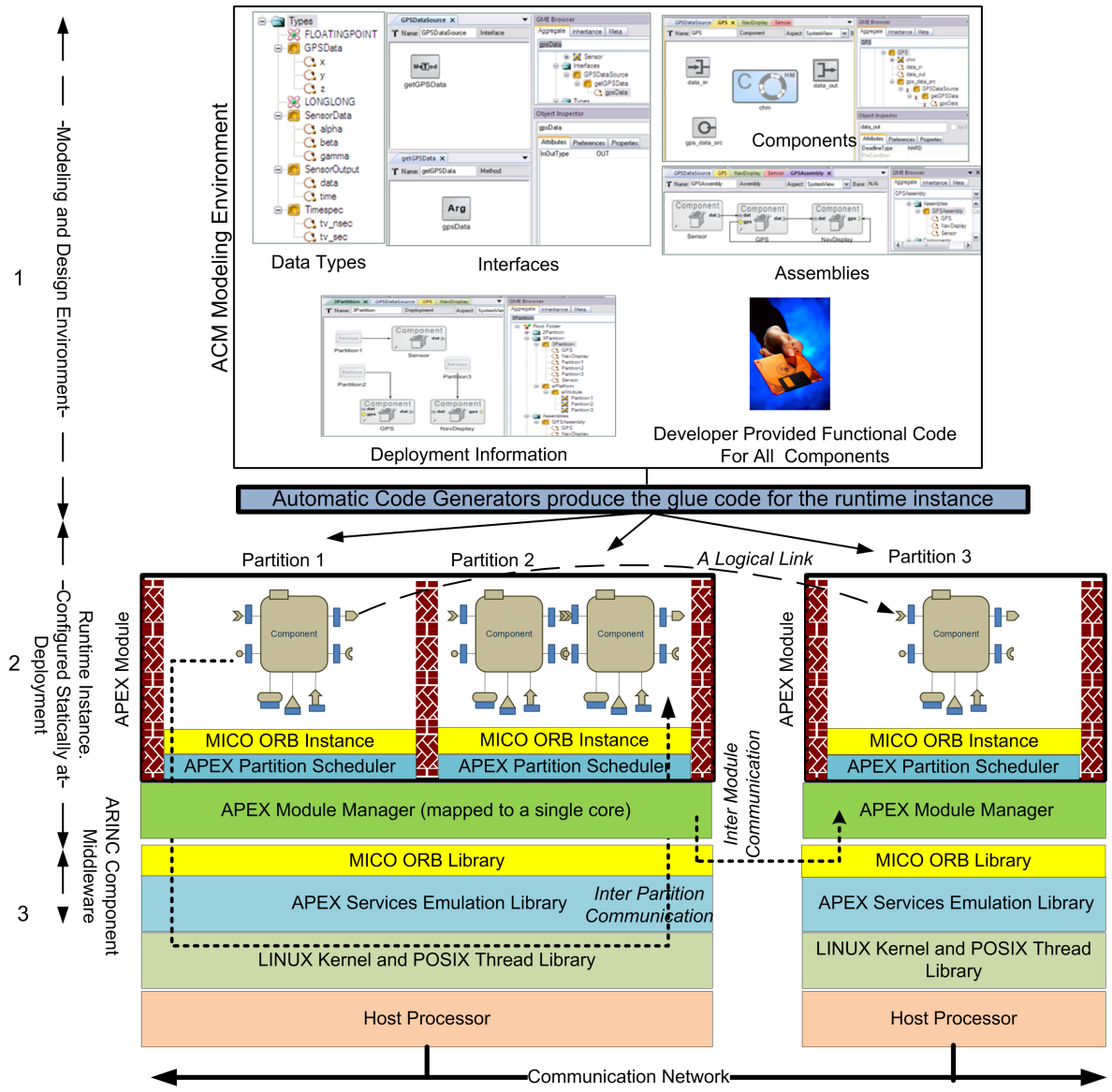


Figure 2: The system architecture, component configuration, component monitoring specifications, assemblies are specified in a domain specific modeling language.

3. (monitored) state variables of the component.

The conditions can be checked either before executing the call (pre-condition) or after executing the call (post-condition).

- **User-provided functional code**: We catch all exceptions in the user provided functional code and abstract it as a violation triggered by a monitor associated with the user code.

The above conditions can be specified via (1) attributes of model elements (e.g. deadline), (2) via a simple expression language (e.g. conditions), or (3) via predefined keywords (e.g. `DATA_VALIDITY`). The expressions can be formed over the (current) values of variables (parameters of the call, or state variables of the component), their *change* since the last invocation, their *rate* of change (change divided by a time value).

Figure 4 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

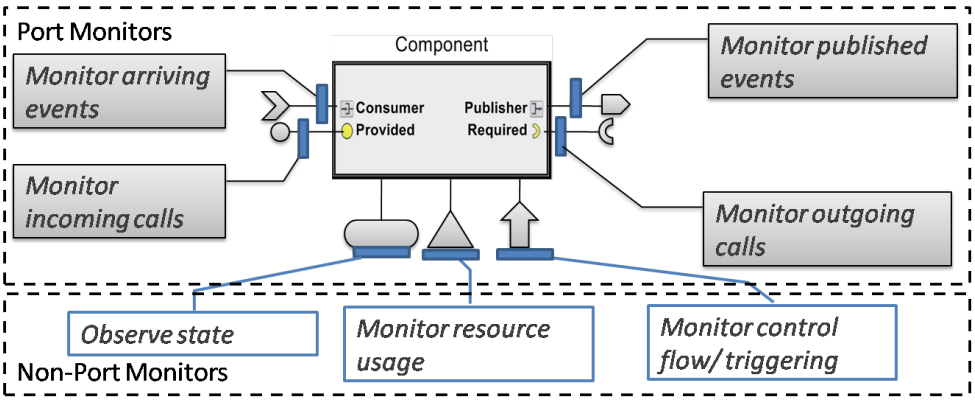


Figure 3: Component Monitoring

always reported to the local component health manager. Deadline violation is always monitored in parallel by the underlying framework. The white boxes are the possible actions taken by the local health manager, and they are discussed in the next section.

4.4 Component-Level Health Management

The monitors observe the conditions specified in the component model. Upon violation, they report the occurrence of the event to the component-level health manager. 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 local 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 include: (see appendix C)
  1. ABORT (also referred to as REFUSE in the text): The faulty process is aborted, it will not finish the current invocation.
  2. IGNORE: The fault is ignored. The faulty process continues the current invocation.
  3. USE PAST DATA: Use the cached value of an incoming data instead of the current value.
  4. STOP: Stops the faulty process permanently, it will not use any more computation resources.
  5. RESTART : It stops the faulty process if it is not already stopped and then restarts it.
  6. RESTORE : This resets the states of a parent component.
- $T : S \times (M \cup Z_{\tau^+}) \rightarrow A \times S$ is the set of all possible transitions that can change the state of the manager due to passage of time or arrival of an input event. To ensure a non-blocking state machine, the framework assumes a default self-transition with IGNORE action if the health manager receives an event which it cannot process in the current state.

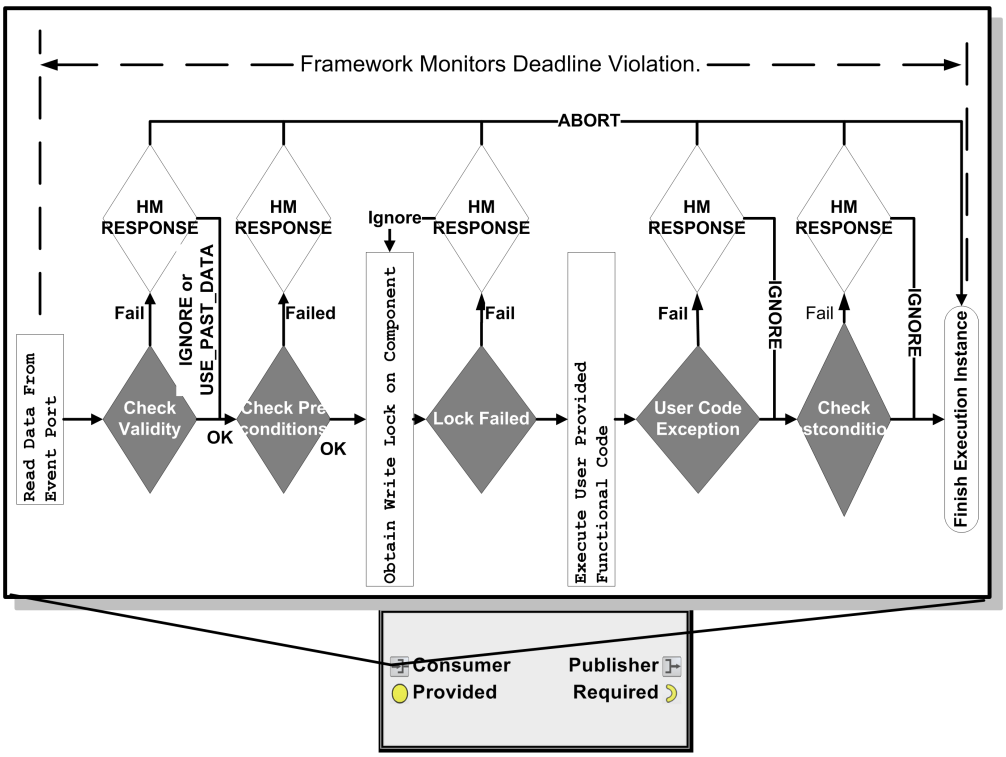


Figure 4: Example of monitors generated for a consumer port.

Figure 5 describes a component level health manager. 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. 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; [1]; they are finally released when the health manager publishes a response (mitigation action) on their respective blackboard.

4.5 System-Level Health Management

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 the fault cascade through the system.

In our implementation, we set up SHM as a separate component which could possibly be deployed in a separate partition or share space in an existing assembly-partition. The SHM component hosts a diagnosis engine that runs in an aperiodic ARINC-653 process. This aperiodic process is triggered either by the alarms published by the monitors (that trigger the CLHM) or by the alarm/response information published by the CLHMs.

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. Please refer to [11, 6] for more details about the TFPG model.

Given the TFPG model of a software assembly, the diagnosis engine reasons about the input alarm and the local response from the component level health manager. It then hypothesizes the possible faults that

[1] Blackboards are primitive inter-process communication structures implemented by ARINC-653 platforms.

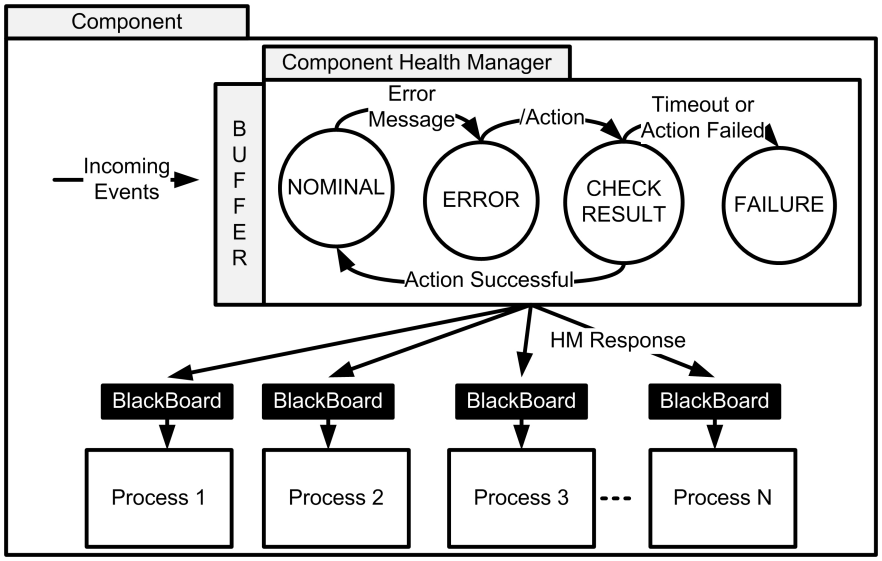


Figure 5: A component level health manager: The state machine inside health manager is application specific and is provided by the developer.

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. System-level mitigation approaches are subject of ongoing investigations.

4.6 Fault Propagation Model

As described in previous sections, in this framework the software assemblies are composed of Components which are in turn composed of specific kinds of ports - Publisher, Consumer, Provides Interface, and Requires Interface. While these interaction ports can be customized by the event-data-types published/consumed, interfaces/methods exposed, periodicity, deadline etc., their fundamental behaviors and interaction patterns are well defined. This implies that it should be possible to identify specific faults and fault propagation pattern that are common to each kind of interaction pattern, which could result in a generic TFPG model for each interaction pattern (connection between two ports of different components). Thus, the failure propagation across the Component boundaries can be captured from the assembly model. The generic TFPG model for a specific interaction port captures the following information:

1. Health Monitor Alarms and the Component Health Manager's Response to these alarms are captured as observable discrepancies
2. Failures originating from within the interaction port and the effect of their propagation as failure modes
3. Effect of failures originating from other entities as discrepancies
4. Cascading effects of failures within the interaction port as discrepancies
5. Effect of failures propagating to other entities as cascades

Additionally, a data and control flow model about the component internals (between the component processes), assists in capturing the failure propagation within the component. In principle, this approach is similar to the failure propagation and transformation calculus described by Wallace [18]. That paper showed how architectural wiring of components and failure behavior of individual components can be used to compute failure properties for the entire system.

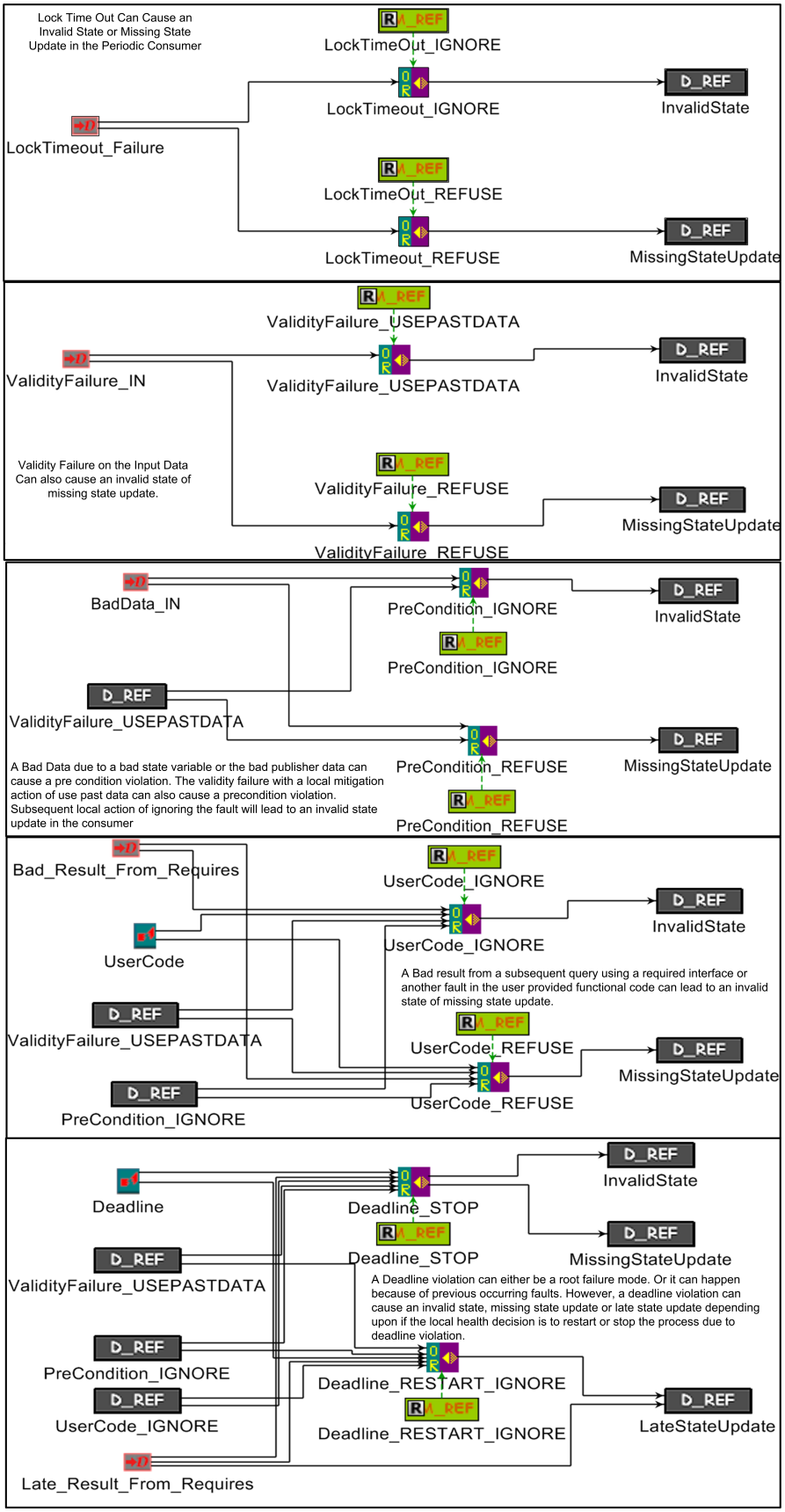


Figure 6: TFPG template showing five (out of six possible) fault propagation patterns for a periodic consumer.

4.6.1 An Example: Generic TFPG for a Periodic Consumer

The figure 6 shows a generic TFPG model for a periodic consumer port. The list below explains the failure effects that are captured in a generic TFPG model through the example of the consumer TFPG model. The Consumer-TFPG model 6, is presented in terms of the failure propagations captured in the context of the observed alarms - LOCK_TIMEOUT, VALIDITY_FAILURE, PRE-CONDITION Violation, USER-CODE Failure, Deadline Violation. Each of these sub-graphs cover most of the points enumerated above.

1. LOCK_TIMEOUT - This is caused by problems in obtaining the Component Lock. Being a real-time system, any attempt to obtain a lock is bounded by a maximum deadline. In case of time out the fault is observed as a discrepancy with an anomaly of LOCK_TIMEOUT, resulting in a CHM response of either IGNORE or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and affect entities downstream.

2. VALIDITY_FAILURE - This is caused when the "age" of the data fed to the consumer is not valid. It is observed as a discrepancy with an anomaly of VALIDITY_FAILURE, resulting in a CHM response of either USE_PAST_DATA or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and effect entities downstream. The failure effect could also cascade into one or more of the anomalies described below.

3. PRE-CONDITION_FAILURE - This is caused when the pre-condition to the consumer process is not satisfied. This anomaly could also be observed as a result of a VALIDITY_FAILURE followed by a response to USE_PAST_DATA. It is observed as a discrepancy with an anomaly of PRE-CONDITION _FAILURE, resulting in a CHM response of either IGNORE or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and affect entities downstream. The failure effect could also cascade into user-code and/or deadline violation.

4. USER-CODE_FAILURE - This is caused when there is a failure in the user-code (e.g. exception). This anomaly could be observed as a result of a USER-CODE failure (captured as a failure mode) or a cascading failure effect propagation from VALIDITY_FAILURE and subsequent health management response of USE_PAST_DATA or PRE-CONDITION violation followed by an IGNORE response. Once observed, depending on the local CHM response its failure effect can lead to an invalid or a missing state update and affect entities downstream. The failure effect could also cascade into a deadline violation.

5. DEADLINE_FAILURE - This is caused when the process deadline is violated. This anomaly could be observed as a result of a Deadline failure (captured as a failure mode) or a cascading failure effect propagation from VALIDITY_FAILURE followed with a local health management response of USE_PAST_DATA or PRE-CONDITION violation with a local CHM response of IGNORE or USER_CODE. It is observed as a discrepancy with an anomaly of DEADLINE_FAILURE, resulting in a CHM response of either STOP (stopping the process in case of hard-deadline violation) or IGNORE (in case of soft-deadline violation) or RESTART. Based on the response, its failure effect can lead to an invalid / missing / late state update and affect entities downstream.

It should be noted that sometimes it might not be possible to monitor some of the failures / alarms mentioned above. In such cases, these observed discrepancies would turn into unobserved discrepancies and the fault effect would propagate through the discrepancy without raising any observation (alarm).

Appendix B lists the timed fault propagation templates for all component ports. These templates are used to construct the TFPG model for the assembly.

4.7 Assembly Failure Propagation Graph

Figure 7 shows the failure propagation link created for interaction between a publisher and a periodic consumer. The publisher and consumer boxes encapsulates the detailed TFPG-model of the publisher and the consumer entities. The failure propagation in-to or out-of the ports captures the failure effect propagation across the entity boundary. Any failure in the Publisher entity that could lead to a discrepancy

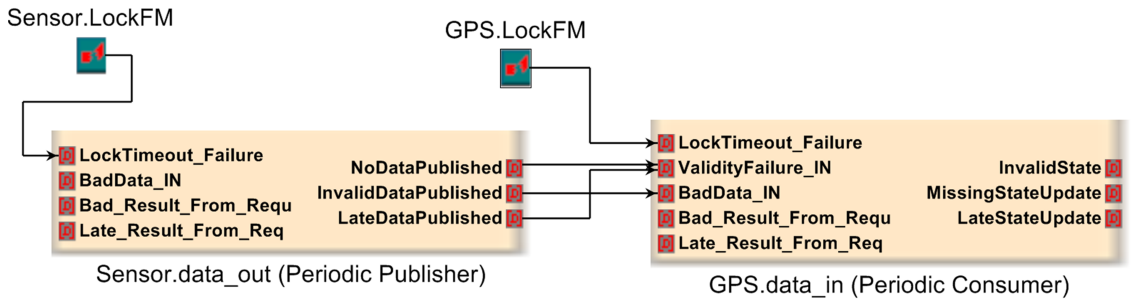
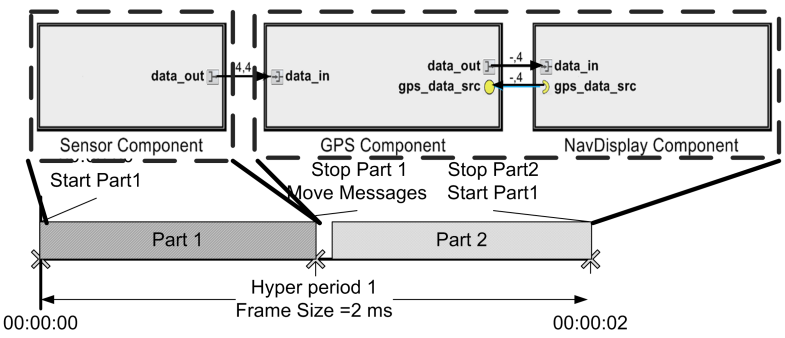


Figure 7: TFPG model for publisher-consumer interaction



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 8: Software Assembly used in the casestudy.

of NoDataPublished / LateDataPublished could possibly cascade to the consumer entity through a VALIDITY_FAILURE in its input data. Likewise, a failure in the publisher leading to InvalidDataPublished discrepancy could produce anomalies in the consumer through the triggering of Bad-Data-Input discrepancy. The model also captures the possibility that a Failure-Mode of a problem in the Component lock could lead to discrepancies associated with Lock_Timeout in the Component's processes/ interaction ports.

Once we can create the models to capture the failure propagation across all interactions, we can essentially create TFPG model for the full assembly. In practice, the assembly level TFPG model is generated by instantiating the appropriate TFPG-model for the Component interaction ports and connecting the failure propagation links between the discrepancy ports. The data and control flow within and across the component can be used to generate the failure propagation links across the instantiated TFPG models. This information can either be obtained by static analysis of the user-level code for the component or relying on the designer to provide this information. Currently, we take the latter approach.

5 Case Study

Figure 5 shows an assembly of three components deployed on two ARINC partitions. Connections between two ports have been annotated with the (periodicity, deadline) 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

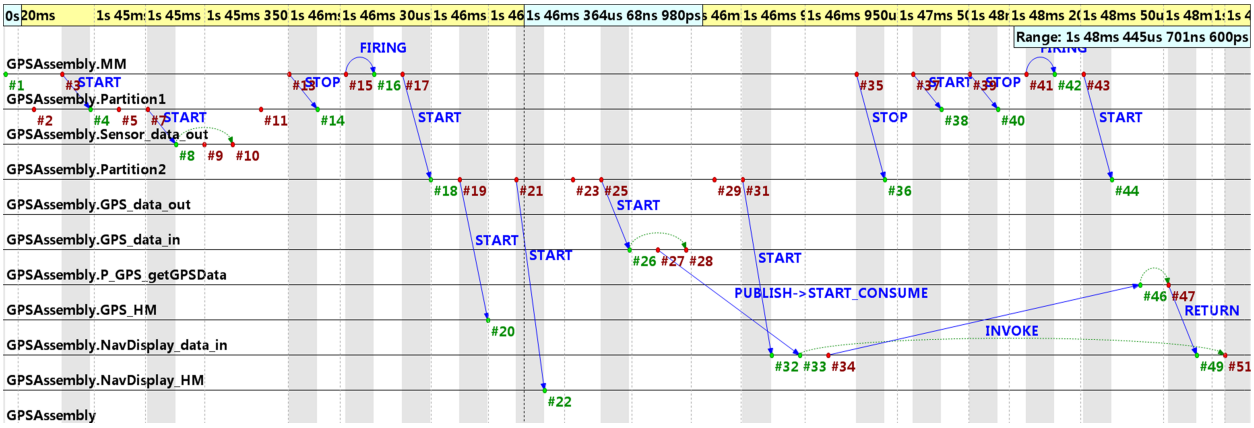


Figure 9: Sequence of Events for a no-fault case. The scale is non-linear.

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.

This figure also describes the periodic schedule followed by the partitions, overseen by a controller process called Module Manager [8]. In this example, Partition 1's phase was 0 milliseconds, while its duration was 1 millisecond. Partition 2's phase was set to 1.0012 millisecond. Its duration was also 0.08 millisecond. The logical 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.

Baseline: No Fault. Figure 9 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++[2]. 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 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 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 5) stops publishing data. First we describe the local component level health management action, which includes local detection as well as mitigation. Then we will see an example of system level diagnosis. System level mitigation has not been included in this example. It is still the subject of ongoing research.

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

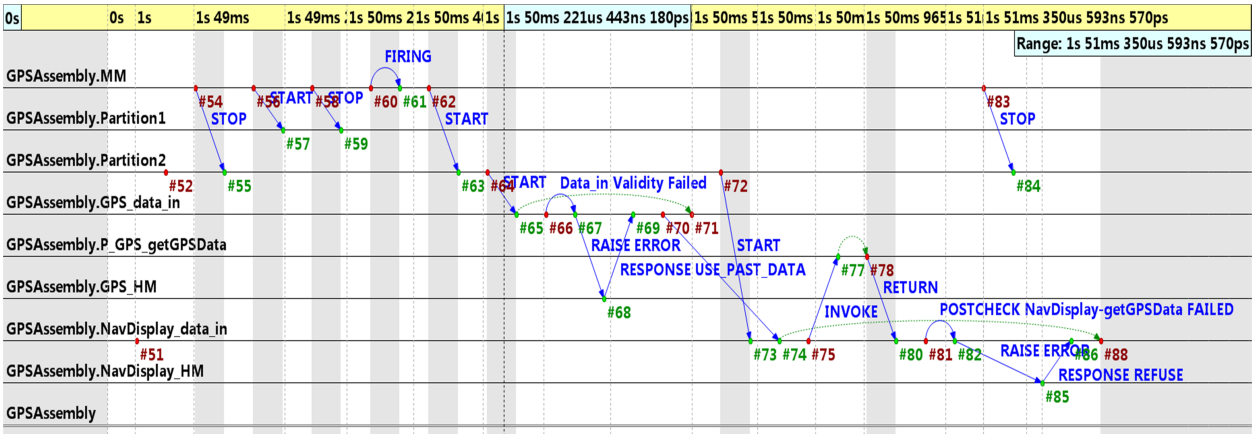


Figure 10: Sequence of Events showing a fault scenario where GPS state is corrupted because the sensor does not update its output as expected. The sensor component is missing from the time line because it does not produce any event. The sequence of events also shows local health management action in the GPS and Nav Display.

5.1 Component Level Health Management Example

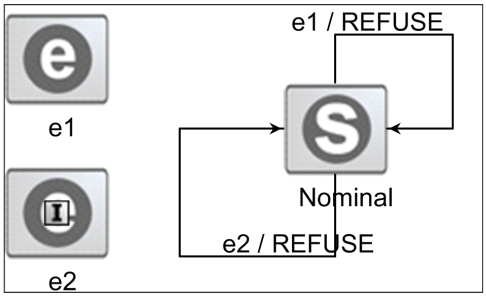
Validity Violation at GPS Consumer Port. Sensor publishes an event every 4 milliseconds in the nominal condition. In this experiment, we injected a fault in the code such the sensor misses all event publications after its first execution. Figure 10 shows the experiment events that elapsed after the sensor fault injection. As can be seen in the figure, there is no activity in Partition1 because of the sensor-fault (event number 57 to 59). The GPS process is started by partition 2 at event 65. At this time, the validity precondition specified in the method that handles the incoming event fails. This precondition 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, which causes the release of GPS health manager. In this case, the GPS health manager (see figure 11) publishes a USE_PAST_DATA response back. The USE_PAST_DATA response 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 results in a fault in the Navigation-Display component. The GPS's getGPSData process sends out bad data when queried by the navigation display. 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. Event numbers 73 to 88 in Figure 10 capture the snapshot corresponding to this experiment. The navigation display component retrieves the current GPS data at event 75 using the remote procedure call. 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 11(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.

Next subsection discusses the system level health management actions related to this fault cascade scenario.

5.2 System Level Health Management Example

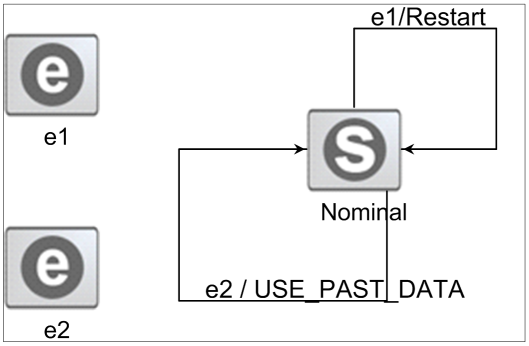
Figure 12 shows the high-level TFPG model for the system/assembly described in figure 5. The detailed TFPG-model specific to each interaction pattern is contained inside the respective TFPG component



e1 (post condition) defined on data_in consumer =(DELTA(data_in.time) > 0)

e2 (post condition) defined on gps_data_source.get_gps_data required interface method =(RATE(gpsData) >1)

(a)



e1 (deadline violation) of the periodic consumer data_in of GPS Component
e2 (Validity Monitoring) defined on the periodic consumer data_in. Validity implies age of data < 4 ms

(b)

Figure 11: (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.

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.

It should be noted that while the interaction pattern 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, the 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. Currently, in the framework there is no instrumentation to deploy a monitor to specifically observe violations in state variable updates. These could be captured indirectly as pre-condition or post-condition monitors on the interfaces/interactions ports that update or read from these state variables.

5.3 System Level Diagnosis Process

Figure 13 shows the assembly in figure 5 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 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 hypothesis, the hypothesis are updated to reflect the refinement in their metrics and alarm list. If the new 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 14 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

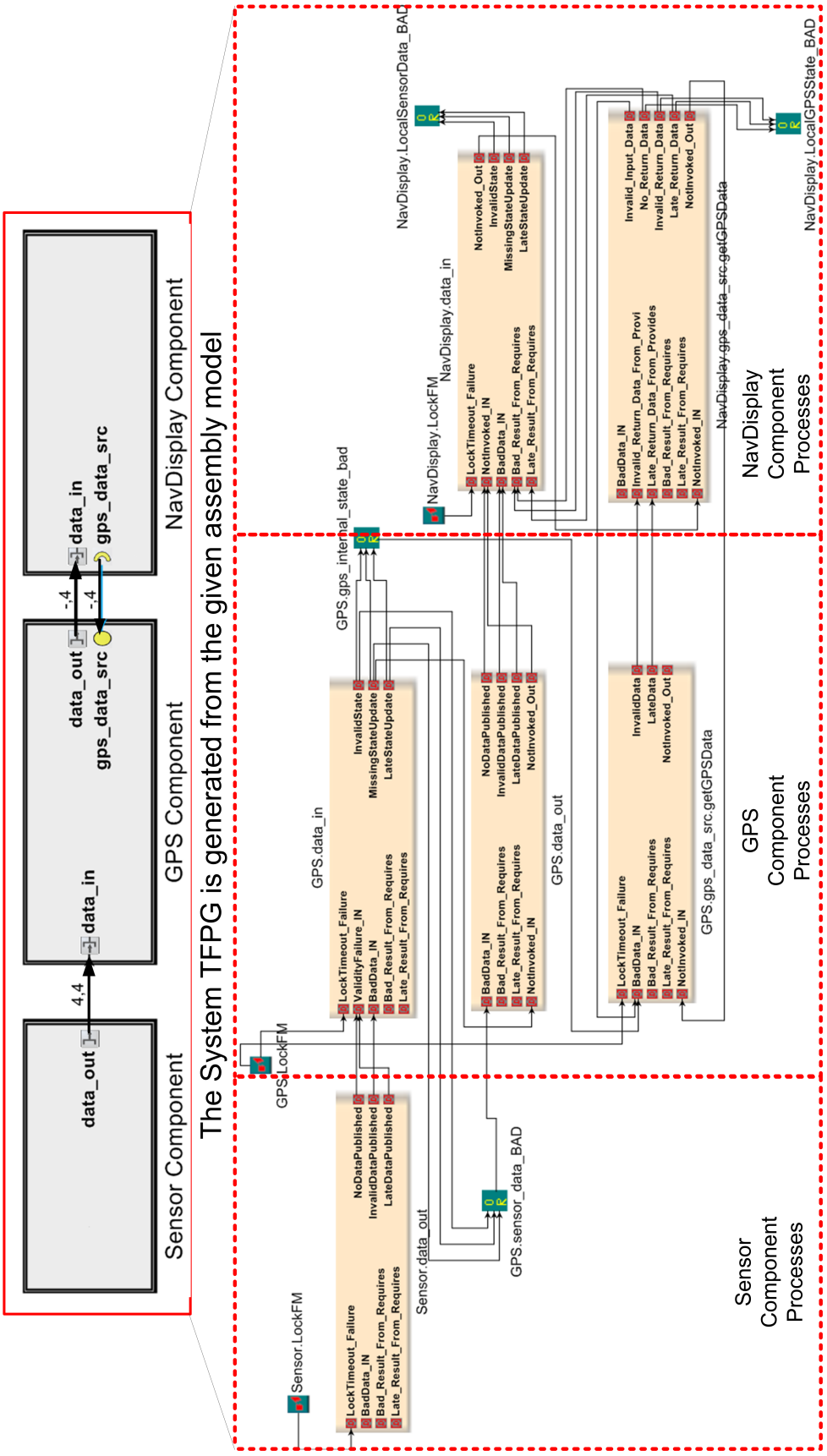


Figure 12: TFPG model for the assembly

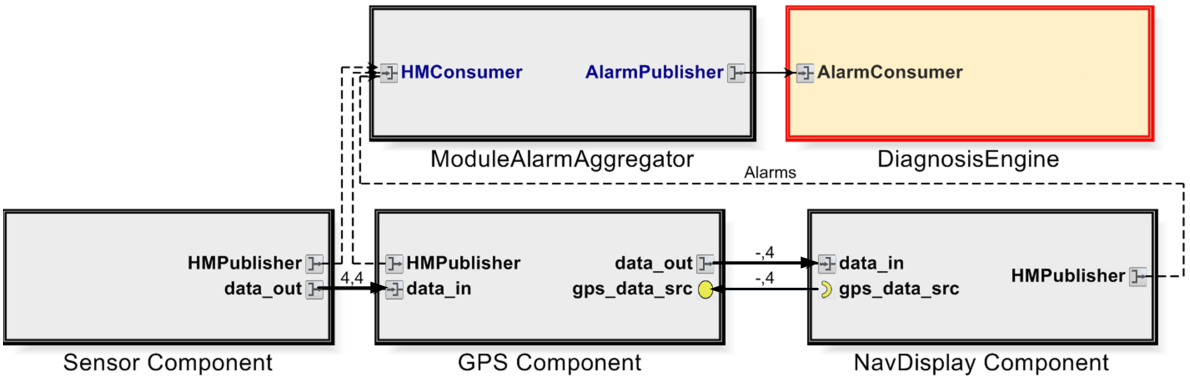


Figure 13: This figures shows augmentation of an assembly with an alarm aggregator component and the diagnosis engine. Compare this to the assembly shown in figure 5.

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.

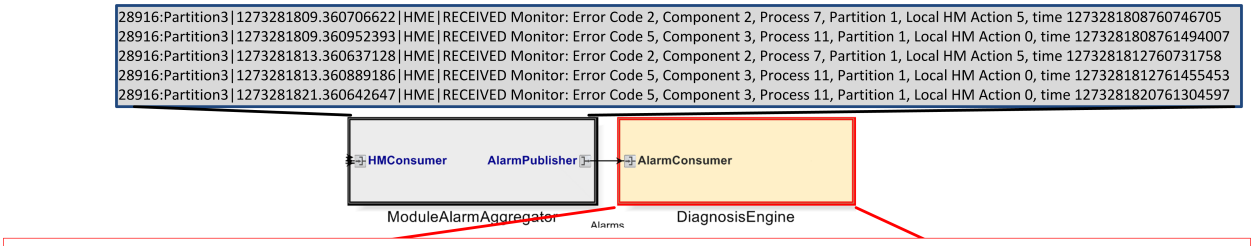
As can be seen from the results, the system correctly generated two hypotheses (figure 14, lines 20 and 24). The first hypothesis blamed the sensor component lock to be the root problem. The second hypothesis blamed the user level code in the sensor publisher process to be the root failure mode. In this situation the second hypothesis was the true cause. However, because we current treat the lock time out monitors as unmonitored discrepancies the diagnoser was not able to reasonably disambiguate between the two possibilities.

6 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 [19] 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 by using the control program to set an appropriate configuration goal that negates 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 [12]. Later this concept was extended to distributed systems by Meyer in [14, 13]. 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 [9].

Conmy et al. presented a framework for certifying Integrated Modular Avionics applications build on ARINC-653 platforms in [7]. 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.




```
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 14: Diagnosis Result from TFPG Reasoner

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 [15]. 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 [10]. 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 [17]. 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.

7 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 focused on building a framework first that combines component-oriented software construction (CCM) 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. Our current work is focusing on extending these concepts to the system level, where faults occur in and propagate across many components, where a more complex diagnosis is needed, and where more sophisticated mitigation logic is necessary. We also plan to extend this work to the entire, larger system: a physical system, like 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.

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

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

[7] Conmy, P., McDermid, J., Nicholson, M.: Safety analysis and certification of open distributed systems. In: International System Safety Conference,. Denver (2002)

[8] Dubey, A., Karsai, G., Kereskenyi, R., Mahadevan, N.: 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)

[9] 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`

[10] Goldberg, A., Horvath, G.: Software fault protection with ARINC 653. In: Proc. IEEE Aerospace Conference. pp. 1–11 (March 2007)

[11] 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)

[12] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)

[13] Jézéquel, J.M., Meyer, B.: Design by contract: The lessons of ariane. Computer 30(1), 129–130 (1997)

[14] Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)

[15] Nicholson, M.: Health monitoring for reconfigurable integrated control systems. Constituents of Modern System safety Thinking. Proceedings of the Thirteenth Safety-critical Systems Symposium. 5, 149–162 (2007)

[16] Puder, A.: MICO: An open source CORBA implementation. IEEE Softw. 21(4), 17–19 (2004)

[17] Sammapun, U., Lee, I., Sokolsky, O.: 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. pp. 147–153 (17–19 Aug 2005)

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

[19] 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)

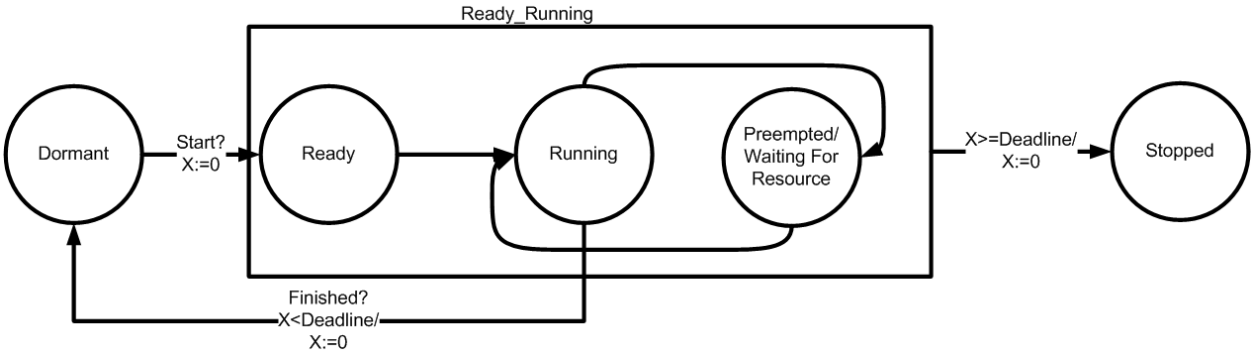


Figure 15: Different States Of An Aperiodic Process

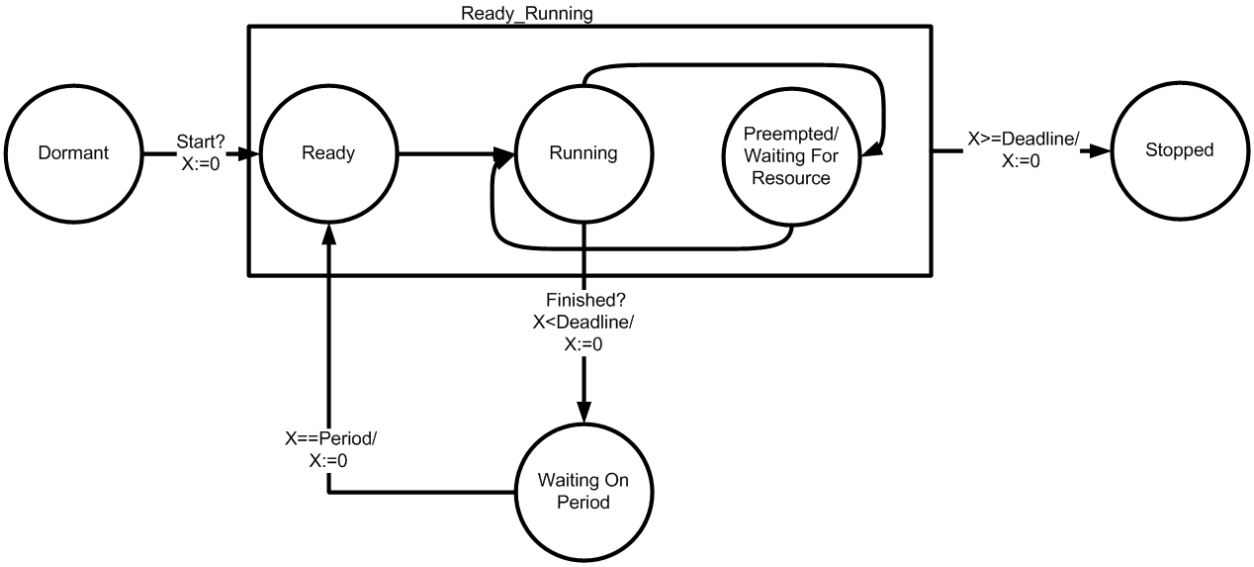


Figure 16: Different States Of Periodic Process

A Execution of Component Ports and Interaction Between Them

All component ports are statically bound to an ARINC-653 process. Periodic publishers or periodic consumers are bound to periodic processes, while aperiodic publishers and aperiodic consumers are bound to aperiodic processes. Each method of a provided or required port is mapped to a unique aperiodic ARINC-653 processes.

Figure 15 shows the timed automaton model for an aperiodic process. Start event is issued by any process in the same partition which wants to release the process. The exact transition from ready to running is unobservable. It happens when the Linux FIFO scheduler picks the process from the process queue and gives it some CPU time. Thereafter, any preemption or wait for resource causes the transition to the waiting state. If the finished condition is true, the process is running and the deadline has not been violated, the process goes back to the dormant state. The partition scheduler is responsible for observing the clock for deadline violation and if necessary moving the process to stopped state. Note that this happens only if the deadline was marked as hard.

Figure 16 shows the timed automaton model for a periodic process. The first start event is issued by the partition upon initialization. The exact transition from ready to running is unobservable. Similar to the aperiodic process, it happens when the Linux FIFO scheduler picks the process from the process queue and gives it some CPU time. Thereafter, any preemption or wait for resource causes the transition to the waiting state. If the finished condition is true, the process is running and the deadline has not been violated, the process goes to the waiting on period state. This is different than the aperiodic process. Thereafter, the

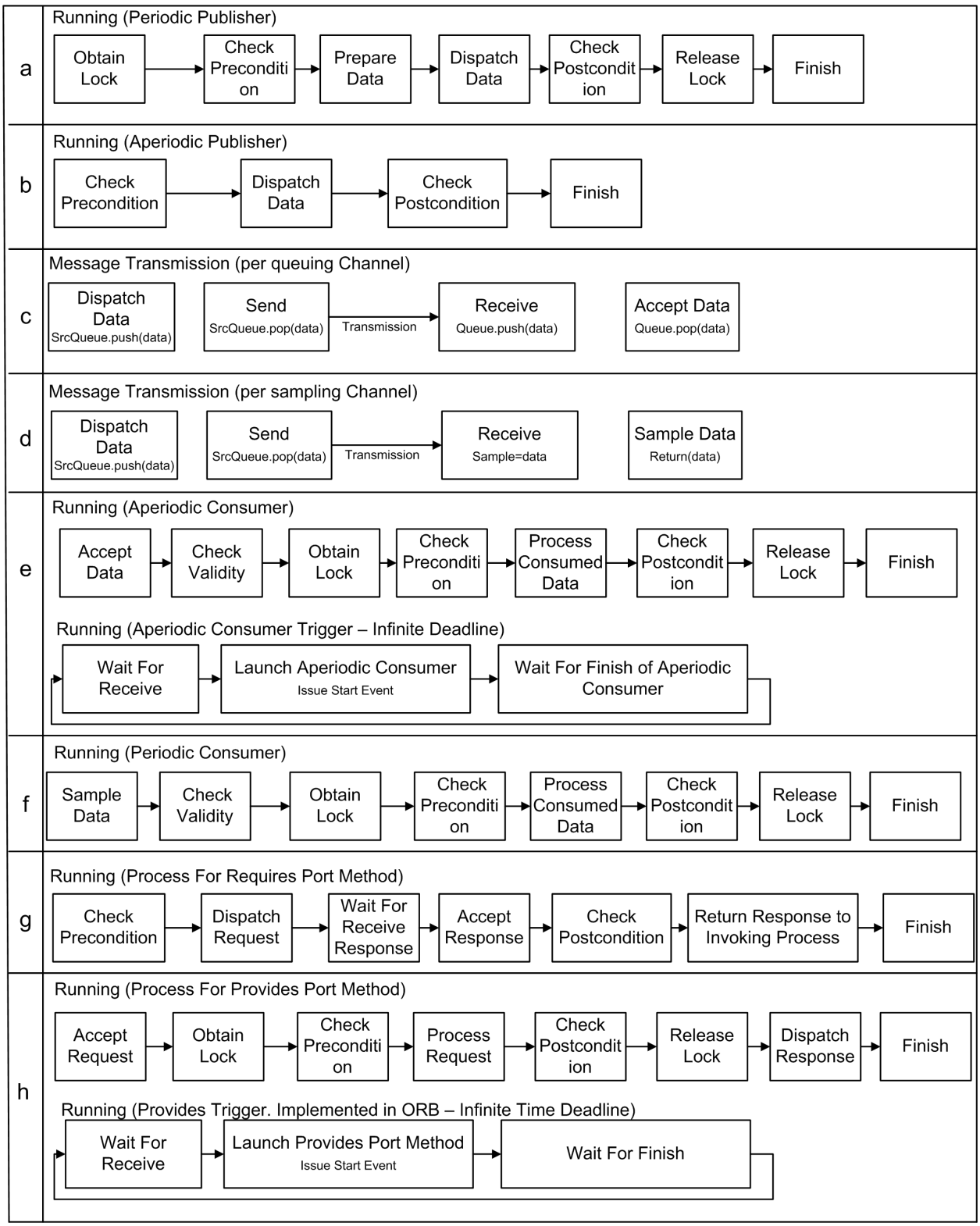


Figure 17: Sequence of Activities Inside the Running State of publishers and consumers and transmission channels

process goes back to the ready state after the expiry of period time. The partition scheduler is responsible for observing and evaluating the clock guards in this automaton.

A.1 Trigger Interaction Pattern

Before we start the discussion on semantics of ports and their interaction it is essential to describe a basic interaction pattern that occurs when one ARINC-653 process invokes another ARINC-653 process. It should be noted that only aperiodic processes can be invoked in this manner.

The invoking process is called the trigger process. This pattern will be seen when we discuss execution semantics of aperiodic consumers and provided ports. Trigger process starts the invoked process by using an API that generates the corresponding `Start` event. Then, the trigger process enters the `waiting for resource` state. Here resource is a semaphore which is reset when the invoked process finishes up. Any processes interacting in this manner must satisfy:

$$Deadline^{Trigger} \geq Deadline^{InvokedProcess} \tag{1}$$

A.2 Ports

Behavior of all ports, periodic or aperiodic is dictated by the sequence of activities that take place when they are in the running state. Figure 17 shows the running states of periodic publishers, aperiodic publishers, periodic consumers, and aperiodic consumers. It also describes the sequence of activities of message transmission.

Below, we explain these activities for each port and explain they interact. We assume the existence of a globally synchronized hyper period across all modules of the software assembly. We also assume the existence of a global clock (T) that is reset at the start of global hyper period and can be used to measure the elapse of time relative to the start of last global hyperiod.

A.3 Periodic Publisher (PP)

Execution of a periodic publisher starts when its ARINC-653 process transitions to the ready state (fig. 16), which happens every $Period^{PP}$ time units. The first execution of the periodic process is delayed by a phase shift, ϕ^{PP}, which is the time when `Start` was called for the process. Thus, the time when the periodic publisher enters the ready state in the K^{th} run of the global hyper period can be written as:

$$T^{PP}_{Ready}(N, K) = N * Period^{PP} + \phi \tag{2}$$

$$N \in \left[0, \frac{Global\ Hyper\ Period}{Period^{PP}}\right) \cap \mathbb{Z} \tag{3}$$

Any valuation of the global clock relative to the start of K^{th} run of the global hyper period can be converted to absolute time since start of the system as:

$$\mathbb{T}(N, K) = T(N, K) + K * GlobalHyperPeriod \tag{4}$$

For brevity, unless we are specifically referring to time evaluations across two different runs of the global hyper periods, we will drop the functional parameter K in the notation.

Once the publisher enters the running state, it attempts to obtain the lock on the component. Then, it checks the preconditions, passes control to the user level code to prepare the data, dispatches the data[3], checks the postconditions, releases the lock and finishes execution. Any failure while obtaining lock, checking preconditions, preparing data (user code), and checking postconditions interrupts the nominal execution and reports the discrepancy to component health manager. Whether execution is continued or aborted depends upon the health manager output. Refer to the main document for more information on health manager. The timed transition trace[4] for nominal execution for periodic publisher is:

[3]Dispatch of data just adds it to network queue. The message transmission happens asynchronously. Look at Figures 17 c and d

[4]A timed transition trace is an execution of the timed automaton. It is written as a sequence of tuples. The first value in the tuple is the event which in this case represents the entry of a state. The second value is typically a vector which contains all clock values. For brevity, we only mention the value of global clock T.

$$< Ready, T^{PP}_{Ready}(N) >, < Lock, T^{PP}_{Lock}(N) >, < PreCond, T^{PP}_{Pre}(N) >, < Prepare, T^{PP}_{Prepare}(N) >,$$
$$< Dispatch, T^{PP}_{Dispatch}(N) >, < PostCond, T^{PP}_{Post}(N) >, < ReleaseLock, T^{PP}_{Release}(N) >,$$
$$< Finish, T^{PP}_{Finish}(N) >$$

$$T^{PP}_{Finish}(N) - T^{PP}_{Ready}(N) \leq Deadline^{PP} \tag{5}$$

A.4 Aperiodic Publisher (AP)

An aperiodic publisher in invoked by some other component process, which generates the `Start` event causing the aperiodic process associated with the AP to transition to ready state. This is the start of execution as shown in figure 15. Upon invocation, the invoking process enters the `Waiting For Resource` state. It goes back to the running state when the invoked aperiodic publisher finishes execution. Given that the invoking process belongs to the same component and is still in `Running` state, unlike periodic publisher, aperiodic publisher does not need to obtain the component lock. Moreover, the data to be published is passed from the invoking process to AP and hence it does not need to spend anytime in the preparing data stage. Rest of its activities are similar to periodic publisher. The timed transition trace for nominal execution is given as:

$$< Ready, T^{AP}_{Ready}(N) >, < PreCond, T^{AP}_{Pre}(N) >, < Dispatch, T^{AP}_{Dispatch} >, < PostCond, T^{AP}_{Post}(N) >,$$
$$< Finish, T^{PP}_{Finish}(N) >$$

Here N is an index that counts the number of time the aperiodic publisher has been executed in the global hyper period.

$$T^{AP}_{Finish}(N) - T^{AP}_{Ready}(N) \leq Deadline^{AP} \tag{6}$$

A.5 Message Transmission (MT)

Ports of components residing in different partitions are connected via channels. These channels[5] can connect one source port to multiple destination ports and are responsible for transmitting the messages.

Figures 17 c and d show the sequence of activities that happens when a message is dispatched from one of the ports. The act of dispatching a message just puts it in a network queue. When the prescribed time comes, the transmission channel `sends` the message over the network. Upon reaching the recipient, the message is `received` and pushed into the application queue , if the receiving port has a queue. Otherwise, the new data overwrites the old data in a sampling port. The timed transition trace for message transmission is given as:

$$< Dispatch, T^{*}_{Dispatch}(-, J) >, < Send, T^{MT}_{Send}(N, K) >< Receive, T^{MT}_{Receive}(N, K) >$$

$$K - J \leq 1 \tag{7}$$

i.e. there can at most one global hyper period between the dispatch of a message and when it is actually sent over the network by the channel. Values of T^{MT}_{Send} can be specified using a time-triggered schedule.

[5] A channel transmitting messages across modules can be implemented as a virtual link in Avionics Full Duplex Switched Ethernet Network. (AFDX)

A.6 Periodic Consumer (PC)

A periodic consumer samples the data it receives via the sampling channel periodically. Like a periodic publisher, the execution of periodic consumer starts when its ARINC process enters the ready state. The first execution of the periodic process is delayed by a phase shift, ϕ^{PC}, which is the time when `Start` was called for the process. The time when execution of periodic consumer starts in the K^{th} run of the global hyper period can be written as:

$$T^{PC}_{Ready}(N,K) = N * Period^{PC} + \phi^{PC} \tag{8}$$

$$N \in \left[0, \frac{Global\ Hyper\ Period}{Period^{PC}}\right) \cap \mathbb{Z} \tag{9}$$

Once PC enters the running state, checks the validity (freshness) of the sampled data. Then it attempts to obtain the lock on the component, checks the preconditions, passes control to user level code to process consumed data, checks postconditions, releases lock and finishes exexution. Any failure in these stages is interrupts the nominal execution and reports the discrepancy to component health manager. The timed transition trace for the N^{th} nominal execution for periodic consumer is:

$$\begin{gathered} < Ready, T^{PC}_{Ready}(N) >, < Sample, T^{PC}_{Sample}(N) >, < Validity, T^{PC}_{Validity}(N) >, < Lock, T^{PC}_{Lock}(N) >, \\ < PreCond, T^{PC}_{Pre}(N) >, < Process, T^{PC}_{Process}(N) >, < PostCond, T^{PC}_{Post}(N) >, \\ < ReleaseLock, T^{PC}_{Release}(N) >, < Finish, T^{PC}_{Finish}(N) > \end{gathered}$$

$$T^{PC}_{Finish}(N) - T^{PC}_{Ready}(N) \leq Deadline^{PC} \tag{10}$$

A.7 Aperiodic Consumer (AC)

Figure 17 (e) describes the various stages that happens when the aperiodic consumer is running. A separate process called aperiodic consumer trigger is responsible for releasing the aperiodic consumer upon receipt of an incoming data. Other difference between periodic consumer and aperiodic consumer is that while the *sample data* activity of periodic consumer is a non-destructive read i.e. the same data item can be read again, the *accept data* activity is a destructive read. Apart from that all other activities of aperiodic consumer are similar to periodic consumer. The timed transition trace for the N^{th} nominal execution for aperiodic consumer is:

$$\begin{gathered} < Ready, T^{AC}_{Ready}(N) >, < Accept, T^{AC}_{Accept}(N) >, < Validity, T^{AC}_{Validity}(N) >, < Lock, T^{AC}_{Lock}(N) >, \\ < PreCond, T^{AC}_{Pre}(N) >, < Process, T^{AC}_{Process}(N) >, < PostCond, T^{AC}_{Post}(N) >, \\ < ReleaseLock, T^{AC}_{Release}(N) >, < Finish, T^{AC}_{Finish}(N) > \end{gathered}$$

$$T^{AC}_{Finish}(N) - T^{AC}_{Ready}(N) \leq Deadline^{AC} \tag{11}$$

A.8 Interaction Between Periodic Consumer and Periodic Publisher or Aperiodic Publisher

The data dispatched by a publisher is a asynchronously consumed by the periodic consumer. The timed trace for the interaction from the point of dispatch to the point of data validity check can be written as follows:

$$\begin{gathered} < Dispatch, T^{*}_{Dispatch}(-,J) >, < Send, T^{MT}_{Send}(-,K) >< Receive, T^{MT}_{Receive}(-,K) >, \\ < Ready, T^{PC}_{Ready}(-,L) >, < Sample, T^{PC}_{Sample}(-,L) >, < Validity, T^{PC}_{Validity}(-.L) > \end{gathered}$$

Here $*$ matches both AP and PP. $-$ for the first index implies that we are referring to the last execution of the process in the specified run of the hyper period. Also,

$$L - J \leq 1 \tag{12}$$
$$L \geq K \geq J \tag{13}$$
$$T^{PC}_{Validity}(-, L) - T^{*}_{Dispatch}(-, J) \leq Validity \tag{14}$$

i.e. there can be atmost one global hyper period difference between dispatch and sampling of the data at the consumer end. And, for nominal run the sampled data's age should be less than the specified validity value.

A.9 Interaction Between Aperiodic Consumer and Periodic/Aperiodic Publisher

The data dispatched by a publisher is a asynchronously consumed by the connected aperiodic consumer. The timed trace for the interaction from the point of dispatch to the point of data validity check can be written as follows:

$$< Dispatch, T^{*}_{Dispatch}(-, J) >, < Send, T^{MT}_{Send}(-, K) >< Receive, T^{MT}_{Receive}(-, K) >,$$
$$< Ready, T^{PC}_{Ready}(-, K) >, < Accept, T^{PC}_{Sample}(-, K) >, < Validity, T^{AC}_{Validity}(-.K) >$$

Here $*$ matches both AP and PP. $-$ for the first index implies that we are referring to the last execution of the process in the specified run of the hyper period. Also,

$$K - J \leq 1 \tag{15}$$
$$T^{AC}_{Validity}(-, K) - T^{*}_{Dispatch}(-, J) \leq Validity \tag{16}$$

i.e. there can be atmost one global hyper period difference between dispatch and sampling of the data at the consumer end. And, for nominal run the sampled data's age should be less than the specified validity value.

Now, we consider execution and interaction semantics of synchronous ports.

A.10 Synchronous (Blocking) Ports

A provided interface port contains the implementation for the methods defined in the provided interface and services the request issued on these interfaces from a connected requires port. Their interaction implies call-return semantics. As described earlier in the main text synchronous ports are always aperiodic. Each method in a synchronous port is executed on its own Aperiodic ARINC-653 Process. The key difference between this interaction from the similar interaction in CCM, is deadline monitoring.

A.10.1 Requires Port (RP)

Like aperiodic publishers, a process for requires port is set to ready state when some other process in the component invokes the corresponding method on the requires port. Figure 17 (g) shows sequence of activities that occur when the requires process is in the running state. It can be noticed that similar to aperiodic publisher, the requires port process does not need obtain a component lock as the invoking process already holds that lock and is blocked till the invoked process is finished. The key difference between aperiodic publisher and the requires port is that after dispatching the request, the requires port process blocks itself and waits to receive the corresponding response from a provides port. Once it receives the response, it accepts it, checks the postconditions, returns the response via shared memory to the invoking process and finishes up.

Note that unlike the interaction between asynchronous ports, communication over synchronous ports results in two pairs of dispatch, send and receive over two different communication channels[6]. The timed transition trace for nominal execution is given as:

$$< Ready, T^{RP}_{Ready}(N) >, < PreCond, T^{RP}_{Pre}(N) >, < DispatchReq, T^{RP}_{DispatchReq}(N) >,$$
$$< Receive, T^{MT}_{Receive} >, < Accept, T^{RP}_{AcceptRes}(N) >, < PostCond, T^{RP}_{Post}(N) >,$$
$$< ReturnResponse, T^{RC}_{RR}(N) >, < Finish, T^{RP}_{Finish}(N) >$$

Here N is an index that counts the number of time the requires port process has been executed in the global hyper period.

$$T^{RP}_{Finish}(N) - T^{RP}_{Ready}(N) \leq Deadline^{RP} \tag{17}$$

Clearly the delay from dispatching the request, transmission of message across the network, processing of request on the provides port, and transmission of response back counts against the deadline of requires port.

A.10.2 Provides Port (PrP)

The behavior of provides port is similar to the behavior of an aperiodic consumer. Here also a trigger process, implemented in the ORB, is responsible for invoking the provides process by issuing the `Start` event when a request is received. If multiple request for the provides port exist, they are queued by at the receives end and serviced in FIFO. Figure 17 (h) shows the sequence of activities inside the running state of a provides port process. Key differences between this port and aperiodic consumer are that provides port process does not check the age of incoming data and that it also acts as an aperiodic publisher and dispatches the response to be transmitted by the communication channel. The timed transition trace for this port can be written as:

$$< Ready, T^{PrP}_{Ready}(N) >, < Accept, T^{PrP}_{Accept}(N) >, < Lock, T^{PrP}_{Lock}(N) >,$$
$$< PreCond, T^{PrP}_{Pre}(N) >, < Process, T^{PrP}_{Process}(N) >, < PostCond, T^{PrP}_{Post}(N) >,$$
$$< ReleaseLock, T^{PrP}_{Release}(N) >, < DispatchResponse, T^{PrP}_{DispatchRes}(N) >, < Finish, T^{PrP}_{Finish}(N) >$$

$$T^{PrP}_{Finish}(N) - T^{PrP}_{Ready}(N) \leq Deadline^{PrP} \tag{18}$$

A.10.3 Interaction Between Provides and Requires Port

The full interaction timed trace for these ports can be written as follows.

$$< Ready, T^{RP}_{Ready}(N,K) >, < PreCond, T^{RP}_{Pre}(N,K) >, < DispatchReq, T^{RP}_{DispatchReq}(N,K) >,$$
$$< Send, T^{MT1}_{Send}(-,K) >< Receive, T^{MT1}_{Receive}(-,K) >,$$
$$< Ready, T^{PrP}_{Ready}(M,K) >, < Accept, T^{PrP}_{Accept}(M,K) >, < Lock, T^{PrP}_{Lock}(M,K) >,$$
$$< PreCond, T^{PrP}_{Pre}(M,K) >, < Process, T^{PrP}_{Process}(M,K) >, < PostCond, T^{PrP}_{Post}(N) >,$$
$$< ReleaseLock, T^{PrP}_{Release}(N) >, < DispatchResponse, T^{PrP}_{DispatchRes}(M,K) >$$
$$< Send, T^{MT2}_{Send}(-,K) >, < Receive, T^{MT2}_{Receive} > (-,K),$$
$$< Accept, T^{RP}_{AcceptRes}(N,K) >, < PostCond, T^{RP}_{Post}(N,K) >, < ReturnResponse, T^{RC}_{RR}(N,K) >,$$
$$< Finish, T^{RP}_{Finish}(N,K) >$$

[6]As described earlier, a communication channel can have only one source, but multiple destinations

$$T^{RP}_{Ready}(N,K) \leq T^{MT1}_{Send}(-,K) \leq T^{MT1}_{Receive}(-,K) \leq T^{MT2}_{Send}(-,K) \leq T^{MT2}_{Receive}(-,K) \tag{19}$$

$$T^{RP}_{Finish}(N) - T^{RP}_{Ready}(N) \leq Deadline^{RP} \tag{20}$$

That is the full interaction between these ports finishes within one global hyper period. And, the timed trigger schedule for message transmission on these channels is such that the cumulative delay encountered in both message transfers is still less than the deadline of the requires port.

Moreover, the deadline of the requires port must be less than the deadline of the process that invoked it, which is required by the constraint of the trigger pattern, see equation 1.

B TFPG Templates for Component Interactions

```
TFPG - Templates
-----------------
The following is a superset of the TFPG entities. Each component port
is populated with a subset of these entities.

1)INDICES
--------

1.1)FAILURE-MODE
--------------
The list of possible failure modes and their associated codes
'LP'-'LOCK_PROBLEM' (is common for all the interfaces in a component)
'VF'-'VALIDITY'
'PREF'-'PRECONDITION'
'UCF'-'USER_CODE'
'POSTF'-'POSTCONDITION'
'DV'-'DEADLINE'

1.2)ALARM
--------------
The list of possible alarms and their codes
'LTF'   -  'LOCK_TIMEOUT_FAILURE'
'VF'    -  'VALIDITY_FAILURE'
'PREF'  -  'PRECONDITION_FAILURE'
'UCF'   -  'USER_CODE_FAILURE'
'POSTF'-  'POSTCONDITION_FAILURE'
'DV'   -   'DEADLINE_VIOLATION'
'SI'   -  'SILENT'

1.3)MODE-VARIABLE
----------------
The list of possible mode variables and their codes
'LTR'-'LOCK_TIMEOUT_RESPONSE'
'VR'-'VALIDITY_RESPONSE'
'PRER'-'PRECONDITION_RESPONSE'
'UCR'  -'USER_CODE_RESPONSE'
'POSTR'-'POSTCONDITION_RESPONSE'
'DVR'-  'DEADLINE_VIOLATION_RESPONSE'
'SIR'  -'SILENT_RESPONSE'

1.4)HM_RESPONSE  (Set of all possible Mode-Values across all Mode-Variables)
---------------
The list of possible responses ( mode-values) from component health manager
'RI'-'IGNORE'
'RA'-'ABORT'
'RS'-'STOP'
'RR'-'RESTART'
'RSU'-'SUSPEND'
'RUPD'  -'USE_PAST_DATA'

1.5)MODE_VALUES per MODE-VARIABLE (The first value is the initial mode)
------------------------------
```

```
The list of possible modes-values per mode-variable
'LTR'   - ['RA','RI']
'VR'    - ['RUPD','RA']
'PRER'  - ['RI','RA']
'UCR'   - ['RI','RA']
'POSTR'- ['RI','RA']
'DVR'   - ['RI','RR','RS']
'SIR'   - ['RI','RA']

1.6)SILENT_OUTPUT_DISCS
--------------------
The list of possible unmonitored output discrepancy ports
'NDP'  - 'NO DATA PUBLISHED'
'LDP'  - 'LATE DATA PUBLISHED'
'IDP'  - 'INVALID DATA PUBLISHED'
'IS'   - 'INVALID STATE'
'LSU'  - 'LATE STATE UPDATE'
'MSU'  - 'MISSING STATE UPDATE'
'IRD'  - 'INVALID RETURN DATA'
'LRD'  - 'LATE RETURN DATA'
'NRD'  - 'NO RETURN DATA'
'IID'  - 'INVALID INPUT DATA'
'NIO'  - 'NOT INVOKED OUT'


1.7)SILENT_INPUT_DISCS
-------------------
The list of possible unmonitored input discrepancy ports.
'LTI'   - 'LOCK_TIME_OUT_FAILURE_IN'
'BDI'   - 'BAD_DATA_IN'
'NDPI'  - 'NO_DATA_PUBLISHED_IN'
'VFI'   - 'VALIDITY_FAILURE_IN'
'IRDPI' - 'INVALID_RETURN_DATA_FROM_PROVIDES_IN'
'LRDPI' - 'LATE_RETURN_DATA_FROM_PROVIDES_IN'
'NII'   - 'NOT INVOKED IN'
'BRFR'  - 'BAD RESULT FROM REQUIRES'
'LRFR'  - 'LATE RESULT FROM REQUIRES'

1.8) DISCREPANCY ASSOCIATED WITH STATE VARIABLES
-----------------------------------------------
Each State variable in the component gets an associated discrepancy
$State_Name + "_BAD_STATE" -

2.)FORWARD PROPAGATION WITHIN INTERFACE
----------------------------------------------
The forward propagation links described below will happen if the source and destination Discrepancy
(monitored / unmonitored) are present within the TFPG associted with the concerned interface.
The forward propagation includes propagation from
- Failure Mode to Monitored Discrepancy (Alarm)
- Un-monitored Input Discrepancy to a Monitored Discrepancy (Alarm)
- Monitored Discrepancy (Alarm)
- one Monitored Discrepancy (Alarm) to another Monitored Discrepancy (Alarm) i.e. Cascade effect
- Un-monitored Input Discrepancy to to Un-monitored Output Discrepancy
```

```
2.1)FORWARD_PROPAGATION (FROM ALARM TO OUTPUT-PORT)
----------------------
For each alarm, the possible down-stream paths within a component is captured.
'LTF'->'NDP'/'NIO'/'MSU'('RA')

'VF'->'IS'('RUPD')
'VF'->'MSU'/'NIO'('RA')

'PREF'->'IDP'/'IS' ('RI')
'PREF'->'NDP'/'MSU'/'NIO' ('RA')
'PREF'->'IRD' ('RI' provides)
'PREF'->'IID' ('RI' requires)
'PREF'->'NRD'('RA' - provides)

'SI'->'MSU'/'NDP'/'NIO'

'UCF'->'IS'/'IDP'/'IRD' ('RI' if UCF is monitored else none)
'UCF'->'MSU'/'NDP'/'NIO'('RA' if UCF is monitored else none)

'POSTF'->'IDP'/'IS'/'IRD'

'DV'->'NDP'/'MSU'/'NRD'/'NIO' ('RS')
'DV'->'LDP'/'LSU' ('RI'/'RR')


2.2)CASCADE-EFFECT WITHIN COMPONENT (ALARM TO ALARM)
---------------------------
'LTF'->'DV' ('RI')
'PREF'->'UCF'('RI')
'PREF'->'POSTF'('RI')
'PREF'->'DV'('RI')
'UCF'->'POSTF'('RI')
'UCF'->'DV'('RI')
'VF'->'PREF'('RUPD')
'VF'->'UCF'('RUPD')
'VF'->'POSTF'('RUPD')

2.3)FORWARD PROPAGATION FROM INPUT-PORT (DISCREPANCY)  TO ALARM
-----------------------------------------
'LTI'->'LTF'
'VFI'->'VF'
'BDI'->'PREF'
'BDI'->'UCF'
'BRFR'->'UCF'
'IRDPI'->'UCF'
'LRDPI'->'UCF'
'BRFR'->'POSTF'
'IRDPI'->'POSTF'
'LRFR'->'DV'
'NII' ->'SI'

2.4)FORWARD PROPAGATION FROM FAILURE-MODE TO ALARM
-----------------------------------
```

'UCF' -> 'UCF'
'POSTF'->'POSTF'
'DV'-> 'DV'

2.5)FORWARD PROPAGATION FROM INPUT-PORT to OUTPUT-PORT

'LRFR'->'LDP'/'LSU'/'LRD'
'NII'->'NIO'/'NDP'/'MSU'/'IS'
'BRFR'->'IID'
'BDI'->'IID'

3)FORWARD PROPAGATION ACROSS INTERFACES

These forward propagation described below happen across interface boundaries (from output port of one to input port of another). While the forward propagation associated with invokes-connection happen within the component, the rest are across component boundaries.

3.1) FAILURE PROPAGATION FROM PUBLISHER TO CONSUMER

'IDP'->'BDI'
'NIO'->'NII'
'NDP'->'NII'
'NDP'->'VFI'
'LDP'->'VFI'
'LDP'->'BDI'

3.2) FAILURE PROPAGATION IN SYNCHRONOUS CALL (FROM REQUIRES TO PROVIDES AND BACK)

'NIO'->'NII' (REQUIRES TO PROVIDES)
'IID'->'BDI' (REQUIRES TO PROVIDES)
'IRD'->'IRDPI' (PROVIDES TO REQUIRES)
'LRD'->'LRDPI' (PROVIDES TO REQUIRES)

3.3) FAILURE PROPAGATION WHILE INVOKING REQUIRES INTERFACE (FROM REQUIRES CLIENT TO REQUIRES AND BACK)

'NIO'->'NII' (REQUIRES-CLIENT TO REQUIRES)
'NRD'->'BRFR' (REQUIRES TO REQUIRES-CLIENT)
'IRD'->'BRFR' (REQUIRES TO REQUIRES-CLIENT)
'LRD'->'LRFR' (REQUIRES TO REQUIRES-CLIENT)

3.4) FAILURE PROPAGATOIN RESULTING FROM INVOKE CONNECTION

'NIO'->'NII' (CLIENT TO SERVER)

4) PROPAGATION WITHIN COMPONENT

This section captures the failure propagations associated with a component.
These include failure propagation from Lock-Problem Failure mode, propagations to and from BAD_State discrepancies.

4.1) FAILURE-MODE

```
-------------------
- Each Component has a failure mode for Lock Problems- 'LTF' FAILURE MODE

4.2) DISCREPANCY (STATE)
--------------------
There is a discrepancy for each state in the component -'_BAD_STATE' DISCREPANCY

4.3) LOCK-TIME-OUT
--------------------
Connect component 'LTF' to 'LTI' of each interface in component.
'LTF'->'LTI'

4.4) BAD_STATE PROPAGATION BASED ON 'READS' CONNECTION
-----------------------------------------------------
CONNECT THE APPROPRIATE _BAD_STATE DISCREPANCY TO BDI
$STATE_NAME + '_BAD_STATE'->'BDI'


4.5) BAD_STATE PROPAGATION BASED ON 'UPDATES' CONNECTION
-------------------------------------------------------
CONNECT TO APPROPRIATE _BAD_STATE DISCREPANCY
'NDP'/'LDP',/IDP'/'IS'/'LSU'/'MSU'/'IRD'/'LRD'/'NRD'->$STATE_NAME + '_BAD_STATE'
```

C Actions of Component Health Manager

The table below provides a brief summary of the effect of each action taken by Component Health Manager.

Table 1: Component Health Manager Actions.

HM 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	Discontinue current operation Aperiodic methods: operation can run again Periodic operations: operation must be enabled by a future START HM action
START	Re-enable a STOP-ped periodic operation
RESET	Stop all operations, initialize state of component, clear all queues, start all periodic operations
CHECKPOINT	Save component state
RESTORE	Restore component state to the last saved state

STOP for all process of a component in combination with start of processes from a redundant component can be used to reconfigure the system. Of course, the channel link from the redundant component should be created at system initialization time.