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

Towards a Real-time Component Framework for Software Health Management

Abhishek Dubey , Gabor Karsai , Robert Kereskenyi , Nagabhushan Mahadevan

TECHNICAL REPORT

ISIS-09-111

November, 2009

Towards a Real-time Component Framework for Software Health Management

Abhishek Dubey Gabor Karsai Robert Kereskenyi Nagabhushan Mahadevan

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

Abstract— **The complexity of software in systems like aerospace vehicles has reached the point where new techniques are needed to ensure system dependability. Such techniques include a novel direction called 'Software Health Management' (SHM) that extends classic software fault tolerance with techniques borrowed from System Health Management. In this paper the initial steps towards building a SHM approach are described that combine component-based software construction with hard real-time operating system platforms. Specifically, the paper discusses how the CORBA Component Model could be combined with the ARINC-653 platform services and the lessons learned from this experiment. The results point towards both extending the CCM as well as revising the ARINC-653.**

I. Introduction

Software today often acts as the ultimate tool to implement functionality in cyber-physical systems and to integrate functions across various subsystems. Consequently, size and complexity of software is growing, often exponentially, and our technologies to ensure that systems are dependable must keep up with this growth. The primary industry practice today is to focus on (1) the extensive testing of systems and subsystems, and (2) carefully managing and documenting the development processes, typically according to some standard like DO-178B [1]. The research practice is focusing on the use of formal (mathematical and algorithmic) techniques, like model checking, static analysis, and abstract interpretation to verify the software. Unfortunately, experience shows that neither approach is perfect and deployed software can still fail, especially under unexpected circumstances. Often such failures arise because of simultaneous hardware failures and latent design (or implementation) errors in the software.

Classic fault-tolerant computing, including software fault-tolerance (SFT) methods [2] do not adequately address these problems. SFT is primarily reactive: when unexpected situations are detected (often in some low-level code) an exception is thrown to an error handler on a higher level, where the fault effect is mitigated (if at all), sometimes without considering system-level effects. Often software fault-management entails rebooting the system (or at least the partition), and fine-grain mitigation is rarely performed. Faults that propagate across the hardware/software interface are poorly understood and managed, leading to problems like the infamous Ariane 5 mishap where a clear discrepancy between the hardware's capabilities and the assumptions made by the software designers led to major loss of property [3]. In the past 10+ years, the new field of System Health Management has been developed [4], especially in the aerospace industry, that deals with detecting anomalies, diagnosing failure sources, and prognosticating future failures in complex systems, like aerospace vehicles. While System Health Management has been developed for physical (hardware) systems, it provides interesting systems engineering techniques for other fields like embedded software systems.

The use of System Health Management techniques for embedded software systems points beyond the capabilities provided by the SFT techniques and can potentially increase a system's dependability. This new direction is called Software Health Management (SHM) [4]. SHM is not a replacement for conventional software- and hardware-fault tolerance, and it is not a replacement for solid engineering analysis, design, implementation, and testing. It is rather an additional capability that applies the concepts and techniques developed for System Health Management for software-intensive systems, at run-time. The main goal of SHM is to prevent a (software) fault from becoming a (system) failure - as in the case of SFT. But, in addition SHM must sense, analyze, and act upon 'software health indicators' that are observable on the running software itself. It also has to provide pertinent information for the operator, to the maintainer, and to the designer of the system.

In this paper, we describe the early results of our project that aims at developing technology for software health management in safety-critical systems. The second section describes a high-level design for software component-level health management; the third section provides background for the real-time component framework we are constructing, while the subsequent sections detail our approach towards combining CORBA Component Model with the hard real-time ARINC-653 platform services and present our results. The paper concludes with a comparison with related work and a summary.

II. High-level design for component level health management

System-level Health Management is an evolving field, but one core engineering assumption is clear for all systems involved: systems are built from components, often called 'Line Replaceable Units' (LRU). In our project, we make a similar assumption: the software itself is built from well-defined, independently developed, verified, and tested components. Components encapsulate (and generalize) objects that provide functionality. They also encapsulate physical device interfaces. Furthermore, all communication and synchronization among components is facilitated by a component framework that provides services for all component interactions. This component framework acts as a middleware, provides composition services, and facilitates all messaging and synchronization among components.

In physical systems, health management can be facilitated on the level of individual components, on the level of subsystems, on the level of system areas, as well as on the global, system level. On the level of components (LRU-s), anomaly detection, fault diagnosis, and prognostics are the typical activities. For example, circuit boards may have built-in fault detection logic, or built-in tests that report to a higher-level health management system. In our project, we first focus on providing software health management on the level of individual software components.

Following the principles of System Health Management, we expect that component-level health management (CLHM) for software components will detect anomalies, will identify and isolate the fault causes of those anomalies (if feasible), will prognosticate future faults, and will mitigate effects of faults. 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 functionality of the CLHM includes various monitoring functions that observe what is happening on the component's interfaces, including monitoring for deadline violations, generate diagnosis and, if possible, prognosis for the health of the component, and take mitigating actions. We believe such functions can be implemented by code that is generated automatically from some higher-level model of the CLHM, and this modeling and generator tool is subject of ongoing research. In this paper, we focus on the following problem only: How can we construct a software component framework that can serve as a foundation for SHM in real-time systems and as a 'platform' for the CLHM? The design of the component framework is heavily influenced by the SHM requirements: the need for monitoring components, diagnosis, prognostics, and mitigation; all in a systematic framework. In the next sections, we first introduce the concepts of our (abstract) component model, and then describe how we implemented a simulated prototype for such a component framework.

III. Background

The two main technologies that we used in this work are the CORBA Component Model and the ARINC-653 platform services. During the course of our research, we had to extend the CORBA Component Model (CCM) to enable SHM. The next two subsections provide the necessary background about the extended component model and the ARINC-653 services.

A. Component Model

Our component model shown in Figure 1 is an extension of the standard CORBA Component Model (CCM). A component is a unit, potentially containing many objects. The component is parameterized, has an externally observable state, it consumes resources, publishes and subscribes to (asynchronous) events, provides interfaces for and requires interfaces from other components that require synchronous interactions. Components can interact via asynchronous/event-triggered and synchronous/call-driven connections. The difference between this model and the standard CCM is the state monitoring and resource interfaces, and the periodic or aperiodic triggering. The health manager can monitor incoming and outgoing calls as well as published and consumed events for anomalies. The state interface allows read-only access to the state of the component, the resource interface allows monitoring the resource usage of the component, and the trigger interface allows monitoring the temporal behavior of the component's methods. We envision that using the resource interface the health manager can detect anomalous resource usage patterns, while on the triggering interface it can detect deadline violations.

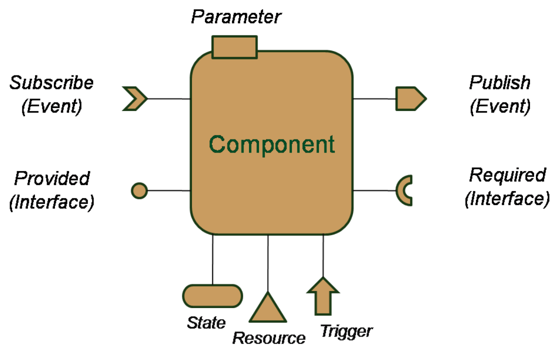


Fig. 1. Component Model

Figure 2 describes an example system built from components. The sampler component is triggered periodically to publish an object containing relevant data. The GPS component is connected to the sampler i.e., it is triggered sporadically to obtain the sampled data. When finished processing, it ends with publishing its own output event. The display component is triggered sporadically to process the GPS event. Upon activation, the display component uses an interface provided by the GPS to retrieve the position data via a synchronous call into the GPS component. Table I gives example periodicity and worst case execution time (WCET) values associated with these tasks.

B. ARINC-653/APEX partitioning kernel

The ARINC-653 software specification describes the standard Application Executive (APEX) kernel and associated services that should be supported by safety-critical real-time operating system (RTOS) used in avionics. It has also been proposed as the standard operating system interface on space missions [5]. The APEX kernel in such systems is required to provide robust spatial and temporal partitioning. The purpose of such partitioning is to provide functional separation between applications for fault-containment. A partition in this environment is similar to an application process in regular

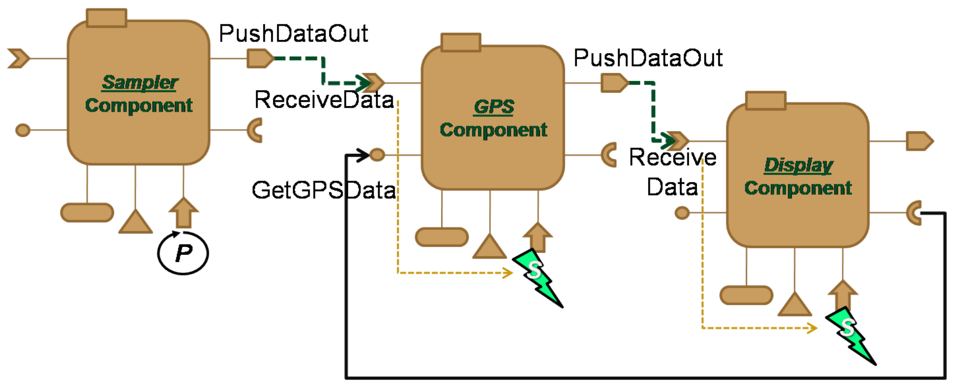


Fig. 2. Example: Component Interactions

TABLE I
GPS EXAMPLE.

Period(secs)	Component	Interface	WCET(secs)
4	Sampler	PushDataOut	4
4	GPS	ReceiveData after processing sends an event to Display	4
Sporadic	Display	ReceiveData after processing calls GetGPSData	4
Sporadic	GPS	GetGPSData	4

operating systems, however, it is completely isolated, both spatially and temporally, from other partitions in the system and it also acts as a fault-containment unit. It also provides a reactive health monitoring service that supports recovery actions by using call-back functions, which are mapped to specific error conditions in configuration tables at the partition/module/system level.

Spatial partitioning [6] ensures exclusive use of a memory region for a partition by an ARINC process (unless otherwise mentioned, a ‘process’ is meant to be understood as an ‘ARINC Process’ in the rest of this paper). It is similar to a thread in regular operating systems. Each partition has predetermined areas of allocated memory and its processes are prohibited from accessing memory outside of the partition’s defined memory area. The protection for memory is enforced by the use of memory management hardware. This guarantees that a faulty process in a partition cannot ruin the data structures of other processes in different partitions. For instance, space partitioning can be used to separate the low-criticality vehicle management components from safety-critical flight control components. Faults in the vehicle management components must not destroy or interfere with the flight control components, and this property could be ensured via the partitioning mechanism.

Temporal partitioning [6] refers to the strict time-slicing of partitions, guaranteeing access for the partitions to the processing resource(s) according to a fixed, periodic schedule. The operating system core (supported by hardware timer devices) is responsible for enforcing the partitioning and managing the individual partitions. The partitions are scheduled on a fixed-time basis, and the order and timing of partitions are defined at configuration time. This provides deterministic scheduling whereby the partitions are allowed to access the processor or other hardware resources for only a predetermined period of time. Temporal partitioning guarantees that a partition has exclusive access to the resources during its assigned time period. It also guarantees that when the predetermined period of execution time of a partition is over, the execution of the partition will be interrupted and the partition itself will be put into a dormant state. Then, the next partition in the schedule order will be granted the right to execution. Note that all shared hardware resources must be managed by the partitioning operating system in order to ensure that control of the resource is relinquished when the time slice for the corresponding partition expires.

IV. IMPLEMENTATION

In this section we describe our implementation of a software component framework that can serve as a foundation for SHM and as a ‘platform’ for the Component Level Health Management (CLHM). Our implementation combines CCM and ARINC-653 as CCM is a suitable starting point for CLHM and ARINC-653 is fit for a software fault-protection system to be used in safety-critical real-time systems.

Figure 3 describes the various layers of our framework called APEXCCM. The main purpose of this framework is to provide support for developing and experimenting with component-based systems using ARINC-653 specifications on top of a Linux operating system. The secondary goal is to be able to design the top layers (component and processes) such that they can be easily rebuilt over standard ARINC-653 compliant operating system.

Hardware Layer: A physical communication network and the physical computing platform form the first two layers.

Operating System: We have selected Linux as the operating system because it is widely available, and it supports a real-time scheduling policy (*SCHED_FIFO*). Moreover, it

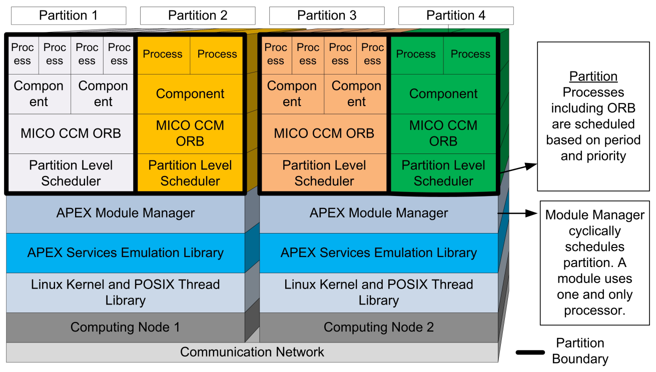

Partition 1
Partition 2
Partition 3
Partition 4
Proc ess
Process
Compon ent
Component
MICO CCM ORB
Partition Level Scheduler
APEX Module Manager
APEX Services Emulation Library
Linux Kernel and POSIX Thread Library
Computing Node 1
Computing Node 2
Communication Network
Partition Processes including ORB are scheduled based on period and priority
Module Manager cyclically schedules partition. A module uses one and only one processor.
Partition Boundary

Fig. 3. Layers of the implemented real-time component framework (APEXCCM).

provides an implementation of the POSIX thread library. In the future, we will explore building our services over the seL4 micro kernel, which has been formally verified with respect to its high-level abstract kernel specification [7]. We rely on the *memory partitioning* between Linux processes provided by the Linux Kernel to implement the spatial partitioning between ARINC-653 partitions in our framework.

APEX Emulation Layer: The next layer consists of our APEX services emulation library. This library provides implementation of ARINC-653 interface specifications for intra-partition process communication that includes Blackboards and Buffers. Buffers provide a queue for passing data messages and Blackboards enable processes to read, write and clear single data message. Intra-partition process synchronization is supported through Semaphores and Events. We have also implemented process and time management services as described in the ARINC-653 specification. Inter-partition communication is provided by sampling ports and queuing ports. We can also provide inter-partition communication using the event channels and remote procedure calls supported by our ORB layer, which will be described later in this section. Overall, this layer was implemented in approximately 15,000 lines of C++ code.

We implement ARINC-653 processes as POSIX threads. ARINC-653 processes, just like POSIX threads share the address space. It should be noted that processes, both periodic and aperiodic, can only be created at initialization, following the ARINC-653 specification. Specified process properties include the expected worst case execution time, which cannot be changed at run-time.

APEX Module Manager: The APEX Module manager forms the next layer in our framework. It is responsible for providing *temporal partitioning* among partitions (i.e., Linux processes). Each partition inside a module is configured with an associated period that identifies the rate of execution. The partition properties also include duration that specifies the time of execution.

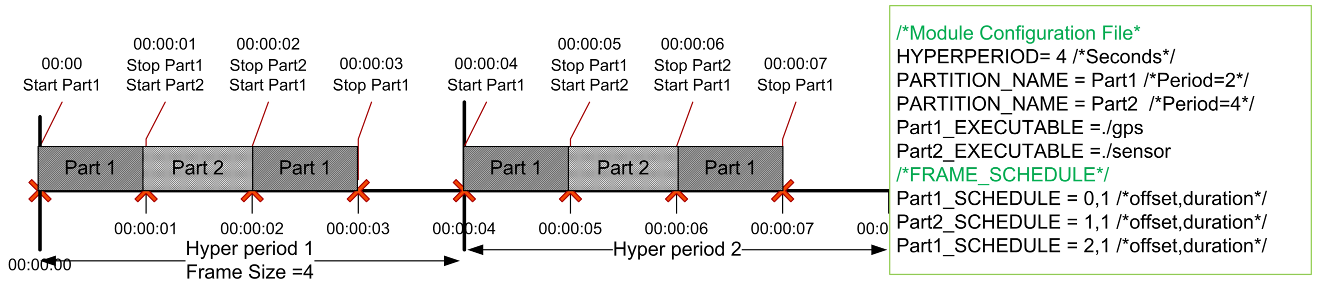
Audsley in [8] and Easwaran in [9] have shown that the periods associated with all partitions in a module should be harmonic i.e., between any given pair of partitions, the period of the first is an integer multiple of the second or vice versa. Moreover, to prevent partition jitter, which may occur if a process wishes to be released when its partition is not executing, the period of the processes inside a partition must be a multiple of the period of the partition.

The module manager is configured with a fixed cyclic schedule that is repeated with a fixed hyperperiod. This schedule is computed from the specified partition periods and durations. The module configuration file also specifies the hyperperiod value, which is the least common multiple of all partition periods, the partition names, the partition executables, and their scheduling windows. Note that the module manager allows execution of one and only one partition inside a specified scheduling window, which is specified with the offset from start of hyperperiod and duration of activity. The module manager is responsible for checking that the schedule is correct before the system can be initialized i.e., given a size 'n' partition schedule tuple $(S_1, S_2, \cdots, S_i, \cdots, S_n)$ sorted in the ascending order of offsets and a hyperperiod value, (schedule instance, $S_i = (PartitionName, Offset_i, Duration_i)$) $(\forall i < n)(Offset_i + Duration_i \leq Offset_{i+1})$ and $Offset_n + Duration_n \leq Hyperperiod$. Figure 4 shows the example execution of a module with two partitions. The hyperperiod is set to 4 seconds in this example.

APEX Partition Scheduler: The next layer is the partition level scheduler. This scheduler is provided by the APEX services emulation library for each partition. It implements a priority-based preemptive scheduling algorithm. This scheduler initializes and schedules the (ARINC-653) processes inside the partition based on their periodicity and priority. It ensures that all processes finish their execution within the specified deadline. Otherwise, a deadline violation is triggered and the process is prevented from further execution, which is the specified default action. It is possible to change this action to allow the restart of processes upon deadline violation.

Object Request Broker (ORB): The next layer is the Object Request Broker (ORB). This framework uses an open source CCM implementation, called MICO [10]. By default, the framework executes the main thread of ORB as an aperiodic ARINC-653 process within the respective partition.

Layers defined by the software developer: The next two layers are provided by the software developer. They include the component definitions and the associated interfaces provided in an IDL file. The software developer is required to supply the interface implementation code, and provide the necessary properties such as periodicity, priority, stack size and deadline. The framework provides the glue code that maps each component interface method to an ARINC-653 process. Due to this mapping, we have to ensure that one and only one instance of a component exists and that the instance is created when the partition is initialized. Also, multiple processes belonging to the same component are automatically configured with a read/write lock depending on whether they have been specified as read-only or not. We will give an example of this mapping and the framework



```
/*Module Configuration File*
HYPERPERIOD= 4 /*Seconds*/
PARTITION_NAME = Part1 /*Period=2*/
PARTITION_NAME = Part2  /*Period=4*/
Part1_EXECUTABLE =./gps
Part2_EXECUTABLE =./sensor
/*FRAME_SCHEDULE*/
Part1_SCHEDULE = 0,1 /*offset,duration*/
Part2_SCHEDULE = 1,1 /*offset,duration*/
Part1_SCHEDULE = 2,1 /*offset,duration*/
```

Fig. 4. An example timeline of events as they occur in a module.

provided code in the next section.

V. Combining ARINC-653 and CCM

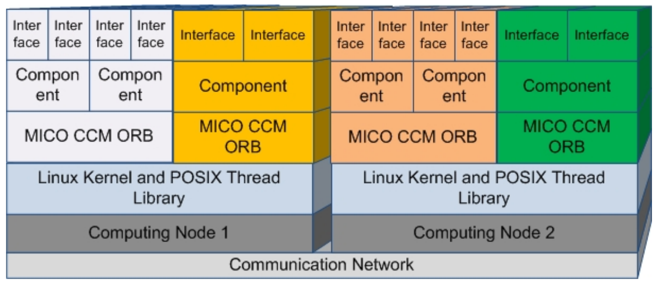


Fig. 5. Typical layers in a CCM implementation

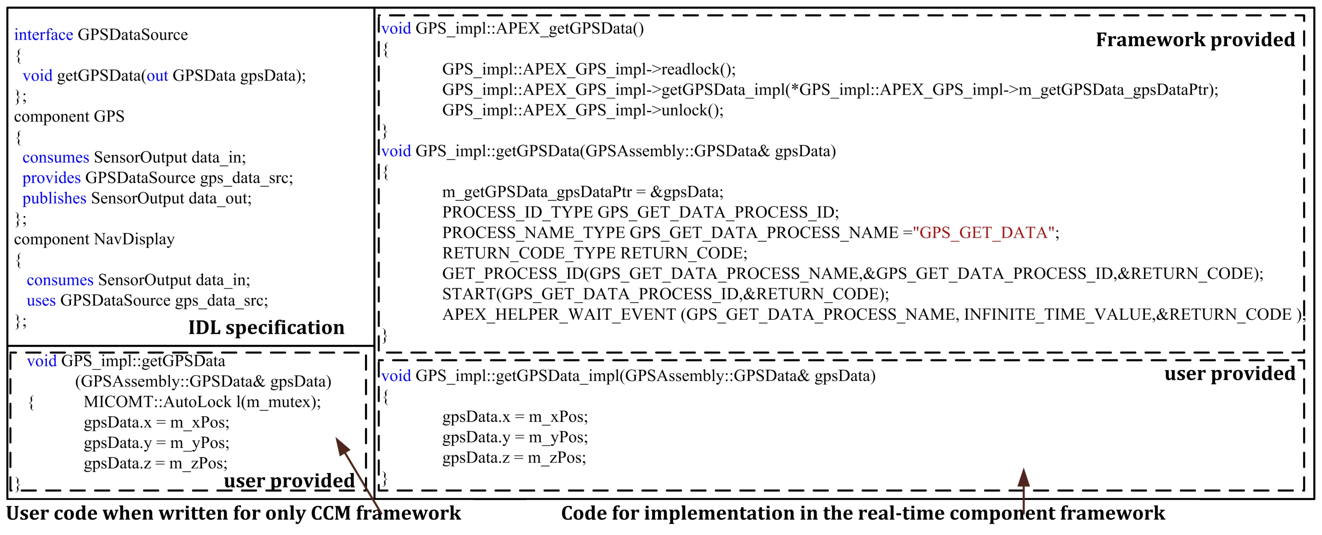
This section highlights some of the key differences in the systems that are ARINC-653 compliant and those that use the CORBA Component Model. Also, we discuss how we combined the two. The section ends with our observations on the restrictions and enhancements that are required in both CCM and ARINC-653 to build a framework that integrates components with a hard-real time platform.

ARINC-653 systems group computational blocks (processes) into Partitions, with one or more Partitions assigned to each Module, and one or more Modules forming a system. The operating system supports both spatial and temporal partitioning. The Partitions and their underlying Processes are created during system initialization. Dynamic creation of Partitions and Processes is not supported. The user configures the Partitions and their underlying Processes with their real-time properties (Priority, Periodicity, Duration, Worst Case Execution Time, Soft/Hard deadline etc.) The Processes are scheduled in real-time and are monitored to check for violations of any real-time constraints. Most often, the Processes are independent of one another and do not share data (except as noted below) and are responsible for their individual state. Intra-partition communication is provided using Buffers that provide a queue for passing data messages and Blackboards that allow processes to read, write and clear single data message. Inter-partition communication is asynchronous and is provided using ports and channels that can be used for sampling and queuing of messages. Inter-process synchronization is supported through Semaphores and Events. ARINC-653 supports a health monitoring service at each layer (Partition/Module/System) which is configured by the user with the appropriate response (functions) for each of the possible faults.

In systems that use the CORBA Component Model (CCM), the computational objects reside inside components that serve as run-time containers and act as a layer between the computational objects and the underlying ORB. Each operating system (OS) process contains an instance of the ORB that hosts the Components as shown in Figure 5. Dynamic memory and dynamic resource allocation are permitted in a typical CCM system. If Components are configured as session-oriented, an instance of the Component is created dynamically for each session request. The methods implemented inside a Component share and contribute to the update of the attributes and the state of the Component. All interactions between Components happen through ports that are used to publish events and to receive subscribed events, or ports that provide or use interface(s). The communication between the objects is achieved through the services provided by the Component layer and the underlying ORB. Additional synchronization support from the ORB service libraries and the underlying OS is also available.

Combining the two: In order to build a Component layer within the ARINC-653 partitions, a mapping needs to be established between the ARINC-653 APEX layer and the CCM layer. The first mapping is done at the physical processor layer. A single physical processor (node) can host multiple partitions in the ARINC-653 domain. In the CCM domain, the same processor can host multiple system/application processes. So at the base level, each ARINC-653 partition is mapped on to a separate application (Linux) process that contains one ORB instance. The ARINC-653 processes within the Partition are mapped to specific threads inside the application (Linux) process. We map each component interface method to a separate ARINC-653 process. This is necessary because the WCET of a process is fixed in the ARINC-653 environment, so one process cannot be used to run interface methods with different WCET-s. Therefore, we had to engineer each interface method to run as a separate ARINC-653 process (implemented as Linux Thread) with the help of some framework-provided code.

Figure 6 shows a portion of the IDL that describes the component example discussed in Section III-A. The bottom left hand side of the figure shows the code written by the user to implement the *getGPSData* interface for the GPS component described in Figure 2, when written for MICO


```
interface GPSDataSource
{
 void getGPSData(out GPSData gpsData);
};
component GPS
{
 consumes SensorOutput data_in;
 provides GPSDataSource gps_data_src;
 publishes SensorOutput data_out;
};
component NavDisplay
{
 consumes SensorOutput data_in;
 uses GPSDataSource gps_data_src;
};
```

```
void GPS_impl::getGPSData
        (GPSAssembly::GPSData& gpsData)
   {    MICOMT::AutoLock l(m_mutex);
        gpsData.x = m_xPos;
        gpsData.y = m_yPos;
        gpsData.z = m_zPos;
}
```

```
void GPS_impl::APEX_getGPSData()
{
        GPS_impl::APEX_GPS_impl->readlock();
        GPS_impl::APEX_GPS_impl->getGPSData_impl(*GPS_impl::APEX_GPS_impl->m_getGPSData_gpsDataPtr);
        GPS_impl::APEX_GPS_impl->unlock();
}
void GPS_impl::getGPSData(GPSAssembly::GPSData& gpsData)
{
        m_getGPSData_gpsDataPtr = &gpsData;
        PROCESS_ID_TYPE GPS_GET_DATA_PROCESS_ID;
        PROCESS_NAME_TYPE GPS_GET_DATA_PROCESS_NAME ="GPS_GET_DATA";
        RETURN_CODE_TYPE RETURN_CODE;
        GET_PROCESS_ID(GPS_GET_DATA_PROCESS_NAME,&GPS_GET_DATA_PROCESS_ID,&RETURN_CODE);
        START(GPS_GET_DATA_PROCESS_ID,&RETURN_CODE);
        APEX_HELPER_WAIT_EVENT (GPS_GET_DATA_PROCESS_NAME, INFINITE_TIME_VALUE,&RETURN_CODE );
}
```

```
void GPS_impl::getGPSData_impl(GPSAssembly::GPSData& gpsData)
{
        gpsData.x = m_xPos;
        gpsData.y = m_yPos;
        gpsData.z = m_zPos;
}
```
IDL specification
user provided
Framework provided
user provided
User code when written for only CCM framework
Code for implementation in the real-time component framework

Fig. 6. Equivalent implementation of a CORBA CCM interface in our framework.

CCM implementation. The right hand side of the figure shows the equivalent code when written in the APEXCCM framework. Notice that the user provided code is the same except that the user is not required to explicitly provide synchronization using locks. The top right corner shows the framework provided code that is used to translate any ORB initiated call to *getGPSData* interface on the GPS component into a start call for the corresponding ARINC-653 process. The generated code also blocks the ORB thread that invoked the CCM method till the corresponding aperiodic process finishes by using the wait call on an APEX event used for notification purposes. We create one notification event per aperiodic process. The framework has to provide component-wide read/write locks to secure the shared access to the component's states among the component interfaces (which are the new ARINC-653 processes).

The main ORB thread in the CCM stack is implemented as an aperiodic ARINC-653 process with an infinite time deadline. Other processes are able to get the processor when the ORB process yields. We restrict the user code from directly launching or creating new threads. This is because a Process in the ARINC-653 environment is not allowed to launch new Processes. Due to the same reason, we had to also configure the ORB to use a predefined number of worker threads (i.e., ARINC-653 Processes) that are created during initialization.

A. *Component Level Health Managers*

The Component Level Health Manager reacts to detected events and takes mitigation actions. It can also report events to higher-level manager (defined at the partition level or the module manager level). In this architecture, the component level health manager is defined as an interface that must be supported by all components, and implemented as a process attached to the component. The framework provides the glue code to create these health-manager processes for the components. They are instantiated as aperiodic processes that run at the highest priority in the partition. The framework

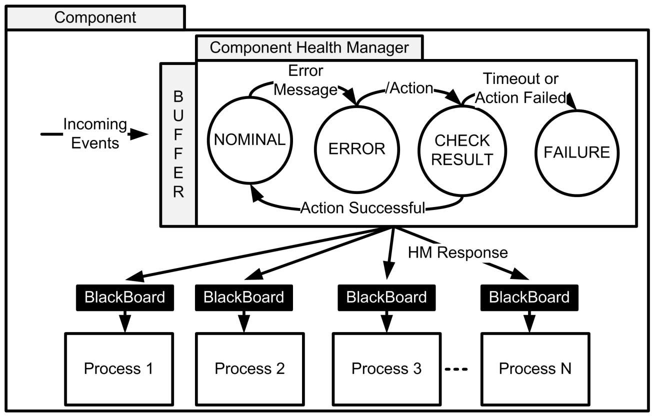


Fig. 7. A component level health manager. The state machine inside health manager is application specific and is provided by the developer.

also provides the glue code for all processes belonging to a component to register with the respective health manager. Upon detection of an error, a component method (process) can use an API to inform the respective health manager that can then take the necessary mitigation section. We will describe the use of health managers using a case study later in this paper.

Figure 7 describes a component level health manager. The process associated with the health manager is sporadically triggered by incoming events that can be generated by either the partition scheduler or other processes registered with the manager. The CLHM's internal state machine tracks the component's state and issues mitigation actions. Processes that that triggered a health management action can block using a blackboard; they are finally released when the health manager publishes a response on their respective blackboard.

In the next sub-sections, we identify some restrictions that had to be enforced (or will be enforced in the future) on the CCM implementation in order for us to be able to combine it with APEX. We also discuss some proposed extensions to

CCM and the lessons that we learned during this process.

B. Restrictions enforced on the CCM implementation

CCM implementations such as MICO are designed for general purpose use. Hence, they allow two kinds of component life-cycles; service and session. While a service component is a singleton, a session component is instantiated for each client request. In an ARINC-653 system, processes cannot be created at run-time. Therefore, we allow only service components i.e., session components are not supported. Moreover, initialization code is provided by the framework to ensure that the component instance is created at the start of a partition.

A related problem is the use of dynamic memory allocation. The ARINC-653 specification requires that all run-time memory allocation should be made on the stack, and not on the heap. Furthermore, in ARINC-653 each process has a specified stack size limit that cannot be violated. To enforce these, the use of memory management hardware is needed.

Recall that we protect the shared access to a Component's state variables through its interface methods implemented as ARINC-653 processes by using a read/write lock. Mixing remote procedure calls provided by the CCM implementation in an ARINC-653 environment can lead to a situation where two or more different processes attempt to acquire the write lock of the same component. This can potentially lead to a deadlock, which will eventually be detected as a deadline violation. To prevent such deadlocks, we require that the call graph of all remote procedure calls be a directed acyclic graph with respect to write lock of all components.

C. Extending CCM

Our component model presented in Section III-A is an extension of the standard CORBA Component Model. We believe that a component level health management system will require interfaces for monitoring resource usage monitoring and deadline violations. Moreover, we propose that the CCM be extended with one health manager per component; a possible improvement over ARINC-653's concept of one health monitor per partition.

The CORBA interceptors could be used to service the health monitors. However, typical CORBA / CCM implementations, including MICO, do not allow the use of request and response interceptors on the client and the server side that are attached to specific Components. However these frameworks allow generic interceptors that are all called for all incoming method calls. An alternative would be to intercept interface specific requests and execute them in the respective component's health manager.

The exception-handling mechanism of the CCM implementation needs to be extended to support resource monitoring and recovery. For example, upon deadline violation, the active Process (thread) must be terminated. However, all locks and resources used by that Process must be released and all other Processes blocked by these locks and resources should be notified. All memory resources should be freed. This service should be made part of the extended CCM specification.

Finally, we also require extensions to the IDL grammar. Currently, this grammar does not support the specification of process attributes such as deadline and periodicity. The extended grammar should allow specification of all ARINC-653 process properties in the IDL. Moreover, we need the ability to define whether or not an interface provided by a component is read-only.

D. Lessons learned

During our experiments as described in the evaluation section, we discovered that in typical CCM implementations like MICO [10] and CIAO [11], the publish-consume connections are implemented as two-way blocking calls and are not really asynchronous. In other words, the publisher's thread will invoke the subscriber's consume methods in the same thread. We are working on implementing an intra-partition event-based communication mechanism for CORBA Components through Blackboards and Buffers provided by our APEX library, where the publisher and the subscriber use separate threads. Also, inter-partition CCM event-based communication will be mapped to sampling and queuing ports.

The ARINC-653 specification stipulates that aperiodic processes are allowed to set or extend their own deadline by using the *replenish()* call, which sets the current deadline to $current\ time + replenish\ time\ request$. Potentially, this can lead to a situation where the current deadline is set to an absolute time, which is earlier than the previous absolute deadline time. This is a potential ambiguity in the specification and should be clarified.

VI. Results

A. Evaluation

We now compare the effort required to develop the GPS example presented in Figure 2 and Table I by using (a) only the APEX services emulation library, and (b) the real-time component framework presented in this paper that provides both ARINC-653 and CCM layers. We did not evaluate the example using only a 'pure' CCM implementation as it does not allow expressing the process periodicity and worst case execution time. In both cases, all services were implemented inside a single ARINC-653 partition.

Using APEX layer: Table I shows the four ARINC-653 processes required to build this example. We implemented the functionality of these processes using the APEX services emulation library that we have built. In order to use the asynchronous event communication, we created two APEX blackboards that were used to store the events communicated between sampler/gps and gps/display respectively. Because of the absence of a component framework, we had to use global variables to store local component state. Consequently, we required three semaphores to protect those variables against race conditions. This example required us, the developers, to write 212 lines of code. Overall, we used APEX API's provided by our library 30 times. Out of those 30 calls, 8 calls were used for Semaphore operations and 6 calls were used for Blackboard operations. There was no framework provided glue code. Since APEX API's

```
////////////////////////////////////////////////
MAXITERATIONS = 10
CPU=1
HYPERPERIOD = 2 //Units: Seconds
PARTITION_NAME = PART1
PARTITION_NAME = PART2
PART1_EXECUTABLE =  ./part1
PART2_EXECUTABLE =  ./part2
PART1_SCHEDULE=  0,1 //Units: Seconds
PART2_SCHEDULE=1,1 //Units: Seconds
PART1_SAMPLINGPORT= SENSOR_SAMPLING_PORT
PART2_SAMPLINGPORT= GPS_SAMPLING_PORT
///////////////////////////////////////////////////////
SENSOR_SAMPLING_PORT_MAXMESSAGESIZE= 1024 //Units: Bytes
SENSOR_SAMPLING_PORT_REFRESHPERIOD= 4  //Units: Seconds
SENSOR_SAMPLING_PORT_DIRECTION= SOURCE
///////////////////////////////////////////////////////
GPS_SAMPLING_PORT_MAXMESSAGESIZE= 1024//Units: Bytes
GPS_SAMPLING_PORT_REFRESHPERIOD= 4 //Units: Seconds
GPS_SAMPLING_PORT_DIRECTION= DESTINATION
////////////////////////////////////////////////////
CHANNEL_NAME= channel1
channel1_SOURCE = SENSOR_SAMPLING_PORT
channel1_DESTINATION = GPS_SAMPLING_PORT
```

Fig. 8. This is the configuration file used by the Module manager for this experiment.

are specified in the standard and are same for all other implementations, this example can be ported to any other ARINC-653 compliant kernel in 212 lines of code.

Using APEXCCM framework: Using the full framework, we were able to take advantage of framework provided code and component abstractions for the same example. Overall, the net APEX API usage was 31 as compared to 30 in the previous case. However, all these invocations were present in the framework provided code, i.e., the software developers were not required to directly invoke any APEX calls. Due to the use of class definitions and declarations, the net lines of code went up to 443. However, only 106 lines of code were provided by the software developer. The rest of the code (337 lines of code) was provided by our framework, which also included the 3 APEX event related calls and 8 component level synchronization calls (we described in earlier sections that we need to protect the common state variables in components using a read/write lock). We did not use Blackboard in this example. Instead, we relied on CCM's publisher/subscriber abstractions. However, as previously mentioned in section V-D we discovered that publish-consume connections are implemented as two-way blocking calls in CCM implementation such as MICO [10] and CIAO [11].

We measured the average jitter of our system to be less than 100 $\mu seconds$ for this experiment, using a vanilla Linux platform running on a dual-core CPU. Most of this jitter can be attributed to the implementation of POSIX condition variables that we use in our library.

B. Component-level software health management demonstration

We deployed the GPS CCM example described earlier in Figure 2 on two ARINC partitions as shown in Figure

```
module HM {
        //Component Health Manager Interface
        interface HealthManager
        {
                //manage the health and do changes
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                void manageHealth(out boolean command);//APERIODIC, WRITE
                void reset();
        };
};//Module Health Manager(HM) ends
module GPSAssembly {
        enum ERROR_CONDITION_CODE_TYPE
        {
                MISSING_GPS_EVENT,
                BAD_GPS_DATA,
                MISSING_SENSOR_EVENT
        };
        enum HM_RESPONSE_TYPE
        {
                CONTINUE,
                BREAK
        };
        //TimeSpec
        struct TimeSpec
        {
                long long tv_sec;  /*sec*/
                long long tv_nsec; /*nono sec*/
        };
        //GPSData
        struct GPSData
        {
                double x,y,z;
        };
        //SensorData
        struct SensorData
        {
                double alpha,beta,gamma;
        };
        //event-token definition used in CORBA-based communication
        eventtype SensorOutput
        {
                public TimeSpec time;
                public SensorData data;
        };
        //APEX equivalent - used in APEX-based communication
        struct APEXSensorOutput
        {
                TimeSpec time;
                SensorData data;
        };
        //interface GPSDataSource
        interface GPSDataSource
        {
                void getGPSData(out GPSData gpsData); //APERIODIC, READ
        };
        //Component Sensor - supports HM::HealthManager
        component Sensor  supports HM::HealthManager
        {
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                publishes SensorOutput data_out; //PERIODIC, WRITE
        };
        //Component GPS - supports HM::HealthManager
        component GPS supports HM::HealthManager
        {
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                consumes SensorOutput data_in; //PERIODIC, WRITE
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                provides GPSDataSource gps_data_src;//APERIODIC, READ
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                publishes SensorOutput data_out;//APERIODIC, WRITE
        };
        //Component NavDisplay - supports HM::HealthManager
        component NavDisplay supports HM::HealthManager
        {
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                consumes SensorOutput data_in;//PERIODIC, WRITE
                //! DEADLINE= , PERIOD = , BASE_PRIORITY= ,
                uses GPSDataSource gps_data_src;//APERIODIC, WRITE
        };
        //CCM Home Definitions
        home SensorHome manages Sensor{ };
        home GPSHome manages GPS {};
        home NavDisplayHome manages NavDisplay {};
};//Module GPSAssembly ends
};
```

Fig. 9. Interface definition used for the GPS example

TABLE II
ARINC PROCESSES CREATED BY THE FRAMEWORK FOR THE GPS EXAMPLE.

Partition	Process Name	Parent Component	Period (seconds)	Time Capacity (seconds)	Deadline Type
Partition 1	Part1 ORB Process	N/A	Aperiodic	Infinite	SOFT
Partition 1	Sensor Process Data Out	Sensor	4	4	HARD
Partition 1	Sensor Health Manager	Sensor	Aperiodic	Infinite	SOFT
Partition 2	Part2 ORB Process	N/A	Aperiodic	Infinite	SOFT
Partition 2	GPS Process Data In	GPS	4	4	HARD
Partition 2	Navigation Process Data In	Navigation Display	Aperiodic	4	HARD
Partition 2	GPS Process Get Data	GPS	Aperiodic	4	HARD
Partition 2	GPS Health Manager	GPS	Aperiodic	Infinite	SOFT
Partition 2	Navigation Health Manager	Navigation Display	Aperiodic	Infinite	SOFT

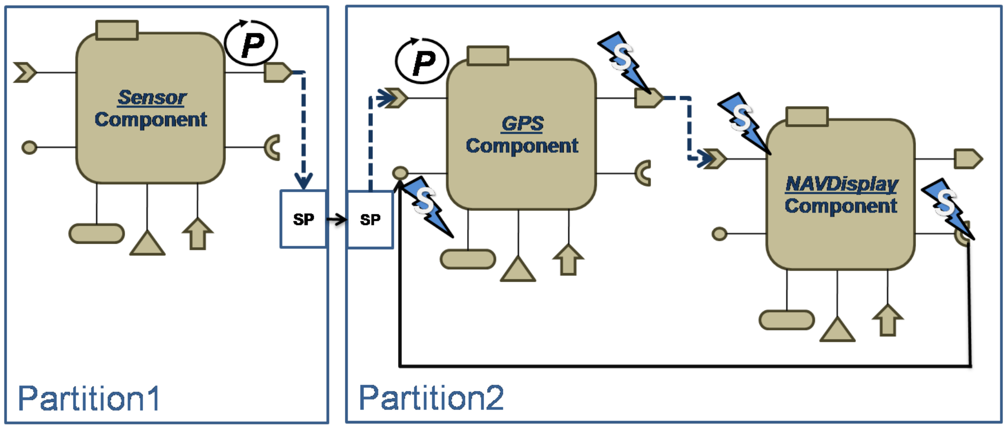


Fig. 10. APEX assembly: We deployed the GPS Example described earlier in Figure 2 on two ARINC partitions. SP is an abbreviation for Sampling Port.

TABLE III
FAULT SCENARIOS.

Fault	Detected at	Fault source	Mitigation
Hard deadline violation	GPS Trigger interface	GPS Component	Stop and restart
Stale data (missing update)	NAVDisplay Subscribe port	GPS Component	Use previous value
Missing sensor event	GPS Subscribe port	Sensor Component	Use previous value
Rate of change is too high	NAVDisplay required interface	GPS Component	Use previous value

10. Partition 1 contains the Sensor Component. The sensor component publishes an event every 4 seconds. Partition 2 contains the GPS and Navigation Display component. The GPS component consumes the event published by sensor at a periodic rate of 4 seconds. Afterwards it publishes an event, which is sporadically consumed by the Navigation Display (abbreviated as display). Thereafter, the display component updated its location by using getGPSData interface provided by the GPS Component. Table II specifies all ARINC processes that were created by the framework for this experiment. Note that the software developer only interacts with the interface methods. All ARINC processes are created and started using the framework generated glue code. Figure 9 shows the corresponding generated IDL file.

Figure 8 describes the experiment configuration parameters. The frame size was set to 2 seconds. Partition 1's phase was 0 seconds, while its duration was 1 second. Partition 2's phase was set to 1 second. Its duration was also 1 second. This ensured that both partition got 1 second of execution time every 2 seconds. Please refer to the description of Module Manager in section IV for an explanation on partition temporal separation. The memory separation is guaranteed by the Linux kernel. Each partition has a sampling port. The Channel connects the source sampling port from partition 1 to destination sampling port in partition 2. The framework is responsible for transferring the messages across a channel from a source port to a destination port (a channel can also link 1 source and multiple destinations together).

We did five experiments with the setup defined in the previous paragraph. The first experiment was designed to measure the baseline performance. It was a no-fault scenario. We then ran four fault scenarios specified in Table III. We describe the results of these experiments in the following sub sections.

Baseline: No Fault Figure 15 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++[1]. 0^{th} event marks the start of the module

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

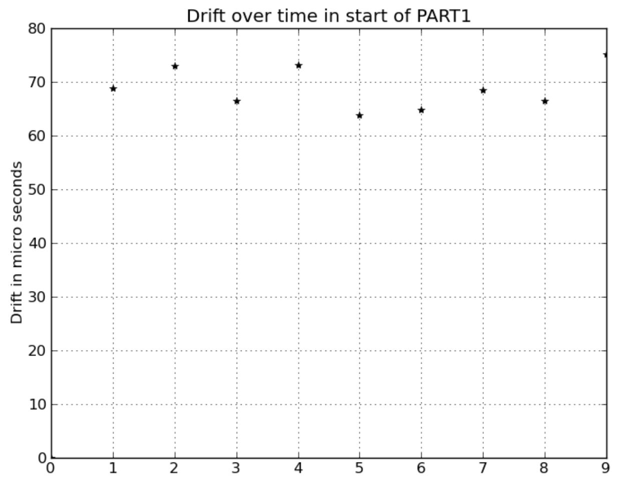


Fig. 11. Absolute jitter (μsec) in execution of Partition 1

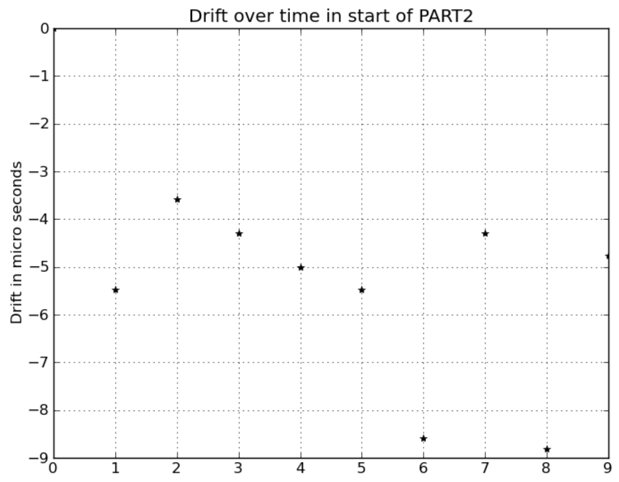


Fig. 13. Absolute jitter (μsec) in execution of Partition 2

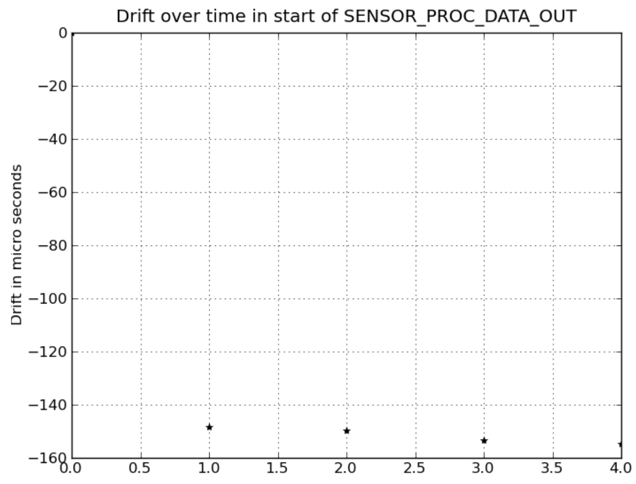


Fig. 12. Absolute jitter (μsec) in execution of Sensor producer process (Sensor Process Data Out)

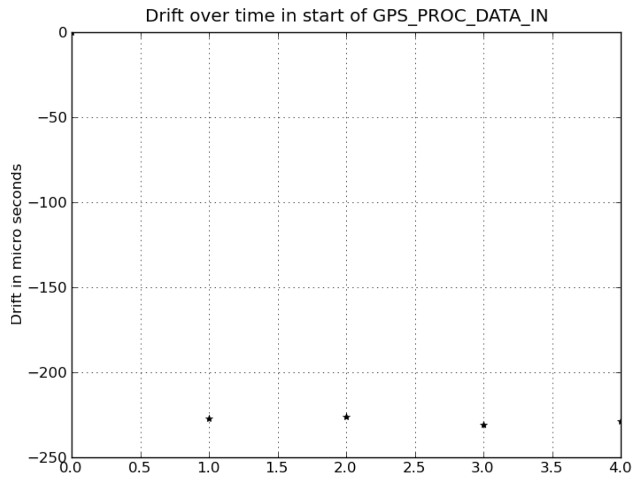


Fig. 14. Absolute jitter (μsec) in execution of GPS consumer process (GPS Process Data In)

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. Approximately, 1 seconds 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 9. The sensor process publishes an event at event number 10 and finishes its execution at event number 11. After 1 second since its start, partition 1 is stopped by the module manager at event number 13. Immediately afterwards, partition 2 is started. Partition 2 starts all its ORB process and health managers at the beginning of its period. At event 23, partition 2 starts the periodic GPS process. It consumes the sensor event at event 24. Notice the cause and effect relationship shown by the arrow. At event 25 GPS process produces an event and finishes its execution cycle. The production of GPS event causes the sporadic release of aperiodic navigation process at event 27. 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 32. It finishes the execution at event 33. Partition 2 is stopped after 1 second from its start at event 35. This marks the end of one frame. There is no sensor and GPS activity in an even frame since the Sensor and GPS periods are 4 seconds and partition periods are 2 seconds.

Figures 11-14 shows the absolute jitter as measured from the start of the experiment. Absolute jitter is defined as the difference between expected release time and actual release of a periodic process.

Fault Scenario: Missing Sensor Event Sensor publishes an event every 4 seconds in the nominal condition. In this experiment, we injected a fault in the code such the sensor misses all event publications between 10 seconds and 20 seconds after its first execution. Figure 18 shows the experiment events that elapsed between 17 seconds and 20 seconds since the start. As can be seen in the figure, the fault is injected in the sensor process at event 144. The GPS process is started by partition 2 at event 152. At this time, the 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 seconds 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 publishes a CONTINUE response back. The CONTINUE response means that the process that detected the fault can continue and use the previously stored data value.

Fault Scenario: Bad GPS Data In this scenario, we inject a fault in the code such that between 10 to 20 seconds since its first launch, the GPS get data process sends out bad data when queried by the navigation display. The bad data is defined by the value of rate of change of GPS data being greater than a threshold. This fault simulates an error in the filtering algorithm in the GPS such that it loses track of the actual position. Figure 17 shows a snapshot of experiment from 18 seconds to 19 seconds. The fault is injected at event 167, approximately 18 seconds after the start of experiment. The navigation display component retrieves the current GPS data at event 176 using the remote procedure call. At event 177, the post condition check of the remote procedure call is violated. This violation is defined by a threshold on the delta change of current GPS data compared to past data (last sample). The navigation display component raises an error at event 179. At event 181 it receives a BREAK response from the health manager. Notice that the execution of navigation display process is preempted till it receives a response from the health manager. The BREAK response means that the process that detected the fault should immediately end 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.

Fault Scenario: Missing GPS Event In this scenario, the GPS fails to update the time stamp in its event properly. This scenario is a combination of two previously described scenarios. It is similar to the missing sensor event as the receiving process (in this case it is the navigation component) does not receive an update. It is similar to the Bad GPS data scenario because the wrong time stamp is a bad piece of data. This fault simulates a case where the GPS clock and the Navigation clock drifts away. Figure 19 describes the sequence of events in this scenario. The fault is injected at event 123. The fault is detected when the precondition associated with the navigation component fails at event 129. This failure is detected by the increased drift or gap between the current time stamp as reported by the navigation component and the time stamp stored in the event. The health manager is informed at event 131. The response given by health manager is BREAK, which causes the navigation process to immediately return. Notice that the navigation process did not issue the remote procedure call to the GPS because its health manager decided that the GPS is faulty.

Fault Scenario: Hard Deadline Violation by GPS Process Data In The final fault scenario illustrates the chain of events that commences when a process misses its hard deadline. In this case, we inject a fault after 10 seconds of the start of the experiment that changes the execution time of GPS component periodic consumer process. The specified deadline time for this process as specified in Table II was 4 seconds. This fault is meant to simulate a situation where an internal equation solver fails to converge to a solution. Figure 16 shows the corresponding sequence of events. The fault is detected at event 110 by the partition 2 scheduler, exactly 4 seconds after the last start of the GPS process. Since the deadline is hard, the partition 2 manager stops the GPS process at event 112 and raises a deadline violation error for that process. This error event is received by the GPS health manager at event 113. In this example, the health manager decides to restart the faulty process. The process restarts at event 115, consumes sensor event at 116, publishes its output and finishes the execution cycle at event 118. The health manager can choose not to restart the faulty process or even reset the component state upon deadline violation. But such mitigation state machines will have to be provided by the software developer.

In this case study, we evaluated the effort required to develop component assemblies in APEXCCM framework. We also described five execution scenarios, one baseline and four fault scenarios. All fault scenarios exhibited basic component-level fault detection and mitigation capability.

VII. Related work

Schedulability analysis for ARINC-653 systems: Audsley et al. presented a discussion on the ARINC-653 standard in [8]. In the same paper, they also presented their work on schedulability analysis of APEX partitions and processes. They showed that partitions can be analyzed in isolation by aggregating the timing requirements of all other partitions together. Work on a similar problem was recently reported by Easwaran et al. [9]. Their work focused on using compositional analysis techniques and took into account the process communication, jitter and preemption overheads. They assumed that all partition periods are harmonic. However, they ignored aperiodic processes. Their techniques can be used to verify the schedule of an ARINC-653 system before deployment.

Lipari and Bini have shown how to compose hierarchical scheduling systems which have a global-level scheduler and a per-application local scheduler [12]. However, they restrict their approach to using a fixed-priority local scheduler. This structure is similar to the one found in an ARINC-653 system. However, in an ARINC-653 system processes are allowed to alter the priority of other processes in the same partition.

Bate and Burns proposed transitioning to a fixed-priority scheduling model from a static priority scheduling model as typically used in the integrated modular avionics systems in [13]. In a priority driven scheduling policy, the priority of the released task decides the execution order, while in the statically set schedule such as that used in scheduling ARINC-653 partition and inter-partition communication, the current time governs the task dispatching order. The problem in time-driven scheduling becomes apparent when one has a distributed system with dependencies across modules and there is drift between corresponding local clocks.

Burns and Lin [14] describe a way to model-check the properties of a single processor real-time system modeled using a constrained form of timed automata. However, their model is restricted due to the semantics of timed automata which does not allow the clock to behave like a stopwatch [15]. Consequently, they can only validate scheduling in systems where each invocation of the task requires some specified computation time and once invoked the task cannot be suspended. This model fits well for schedulability analysis of partitions in ARINC-653 systems which are statically scheduled. However, it cannot be used for analyzing ARINC-653 processes which can be suspended during execution by other processes.

These analysis algorithms require the knowledge of the worst case execution time associated with each task. However, estimating worst case execution times is difficult and as a consequence it is possible that deadlines are violated during run-time. Therefore, the run-time deadline violation monitoring provided by frameworks such as ours ensures that the assumptions made about schedules at design time remain valid even if one or more process in a partition are faulty.

Related frameworks: An approach to objects based on time-triggered (periodic) and event-triggered (sporadic) methods has been presented in [16]. The approach described is implemented in the form of object structures, and many concepts are similar to our approach. However, there are two differences: we rely on an industry standard (ARINC-653) as the underlying platform, and we build a framework on top of that to provide specific services for component interactions and scheduling (for SHM).

Kuz et al. presented a component model called CAmkES in [17]. They built their system above the L4 micro kernel. CAmkES does not provide temporal partitioning. Instead, it is designed to be a low-overhead system that can run on small computing nodes by enforcing static components (i.e., a singleton and not a session-based component) and static bindings. We had to also enforce similar restrictions in our framework to keep the component interactions simple and predictable. While this framework has been built and tested on ARM processors, our prototype ARINC-653 and CCM framework has been developed for the x86 architecture as it gives us more flexibility in experimenting with the SHM technology.

Delange et al. recently published their work on POK (PolyORB Kernel) [18]. It uses AADL specifications to automatically configure and deploy processes and partitions to a QEMU based emulated computing node. We are currently working on obtaining details about this project. DIANA [19] is a new project for implementing an avionics platform called Architecture for Independent Distributed Avionics (AIDA) using Java as the core technology. One of the challenges in using Java is the threading model, which requires their Java virtual machine called PERC Pico to handle the thread scheduling itself instead of the operating system. This adds another layer of scheduling above the operating system. Hence, they do not provide a one-to-one mapping between a Java thread and an APEX process. Another issue with using Java mentioned in their paper is the complexity in estimating and bounding memory usage per thread, which is a critical requirement in the ARINC-653 standard. Finally, they also mention that the errors signaled by PERC Pico are hard to diagnose and correct [19]. This is partially due to the extra layer imposed by the Java virtual machine.

Lakshmanan and Rajkumar presented a distributed resource kernel framework used to deploy real-time applications with timing deadlines and resource isolation in [20]. Their system consists of a ‘partitioned’ virtual container built over their Linux/RK platform. They have reported that their framework provides temporal resource isolation in that they ensure that the timing guarantees provided to each independent application do hold irrespective of the behavior of other applications by using CPU as a reserved resource. However, to the best of our knowledge they do not support process and partition management services as specified in ARINC-653. Moreover, their framework does not support a component model.

Fault detection and health management: Conmy et al. presented a framework for certifying Integrated Modular Avionics applications build on ARINC-653 platforms in [21]. Their main approach was the use of ‘safety contracts’ to validate the system at design time. They defined the relationship between two or more components within a safety critical system. However, they did not present any details on the nature of these contracts and how they can be specified. We believe that a similar approach can be taken to formulate acceptance criteria, in terms of “correct” value-domain and temporal-domain properties that will let us detect any deviation in a component’s behavior.

Mark Nicholson presented the concept of reconfiguration in integrated modular systems running on operating systems that provide robust spatial and temporal partitioning in [22]. He identified that health monitoring is critical for a safety-critical software system and that in the future it will be necessary to trade-off redundancy based fault tolerance for the ability of “reconfiguration on failure” while still operational. He described that a possibility for achieving this goal is to use a set-of look up 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. Finally, he identified the challenge of validating systems that can reconfigure at run-time.

Goldberg and Horvath have discussed discrepancy monitoring in the context of ARINC-653 health-management architecture in [6]. 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 [23]. 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. The 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.

VIII. Conclusions

This paper presented our first steps towards building a Software Health Management technology that extends beyond classic 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). We created a prototype using Linux processes and POSIX threads for purely experimental purposes. However, the principles and techniques developed are portable to 'real' ARINC-653 implementations. During this effort, we have recognized several discrepancies between CCM and ARINC-653, and these differences lead us to believe that further developments are needed that integrate components with a hard real-time platform.

IX. 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 Paul Miner, Eric Cooper, and Suzette Person of NASA Langley Research Center for their help and guidance on the project.

References

[1] "DO-178B, Software considerations in airborne systems and equipment certification," RTCA, Incorporated, RTCA, Incorporated, 1992.

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

[3] "Ariane 5 inquiry board report," Available at http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, Tech. Rep., June 1996. [Online]. Available: http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[4] "NASA IVHM technical plan," Available for download at www.aeronautics.nasa.gov/nra_pdf/ivhm_tech_plan_c1.pdf.

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

[6] A. Goldberg and G. Horvath, "Software fault protection with ARINC 653," in *Proc. IEEE Aerospace Conference*, March 2007, pp. 1–11. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4161684

[7] G. Klein, K. Elphinstone *et al.*, "seL4: Formal verification of an OS kernel," in *SOSP 2009: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, Montana: ACM, October 2009.

[8] N. Audsley and A. Wellings, "Analysing APEX applications," in *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1996, p. 39.

[9] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional framework for avionics (ARINC-653) systems," University of Pennsylvania, Technical Report MS-CIS-09-04, February 2009.

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

[11] "CIAO - component integrated ACE ORB," Available for download at http://download.dre.vanderbilt.edu/.

[12] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, no. 2, pp. 257–269, 2005.

[13] I. J. Bate and A. P. Burns, "An integrated approach to scheduling in safety-critical embedded control systems," *Real-Time Systems*, vol. 25, no. 1, pp. 5–37, 2003.

[14] A. P. Burns and T. M. Lin, "An engineering process for the verification of real-time systems," *Formal Aspects of Computing*, vol. 19, no. 1, pp. 111–136, March 2007. [Online]. Available: http://dx.doi.org/10.1007/s00165-006-0021-4

[15] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[16] K. Kim, "Object structures for real-time systems and simulators," *Computer*, vol. 30, no. 8, pp. 62–70, Aug 1997.

[17] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007.

[18] J. Delange, L. Pautet, and P. Feiler, "Validating safety and security requirements for partitioned architectures," in *Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*. Berlin, Heidelberg: Springer-Verlag, June 2009, pp. 30–43.

[19] T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier, and M. Richard-Foy, "Use of PERC Pico in the AIDA avionics platform," in *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. New York, NY, USA: ACM, 2009, pp. 169–178.

[20] K. Lakshmanan and R. Rajkumar, "Distributed resource kernels: OS support for end-to-end resource isolation," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 195–204, 2008.

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

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

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

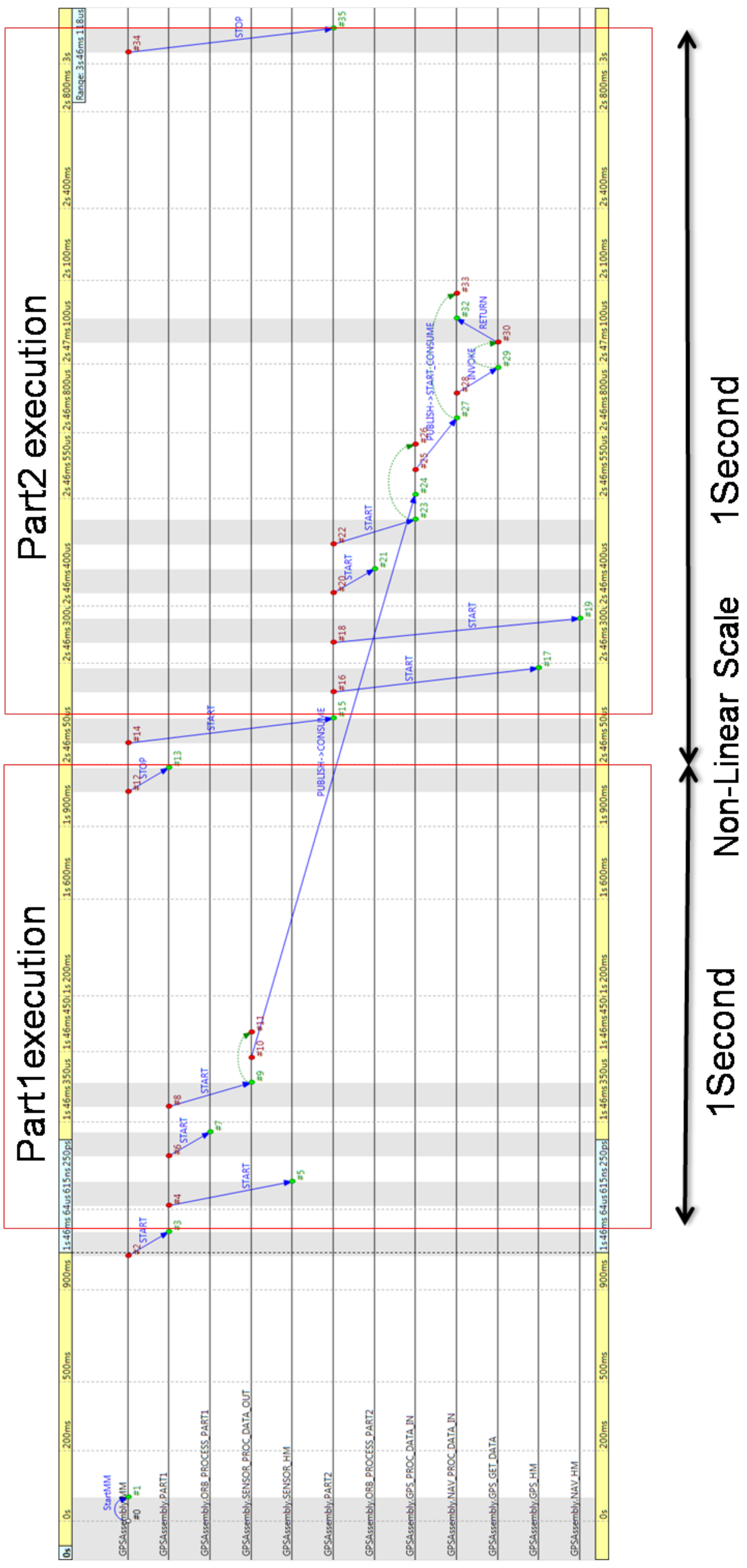


Fig. 15. Sequence of Events for a no-fault case

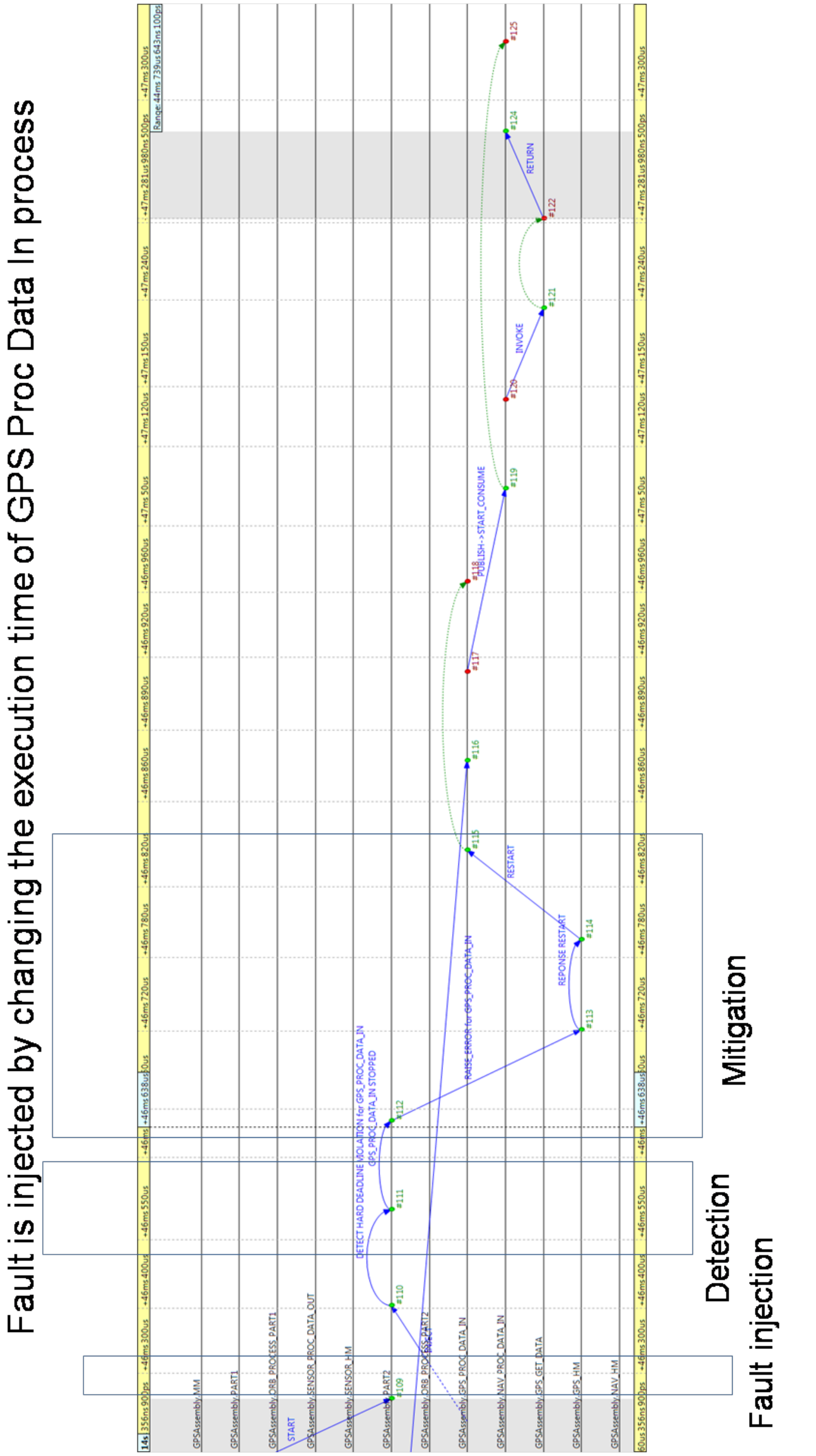


Fig. 16. Sequence of Events for a GPSDeadline case

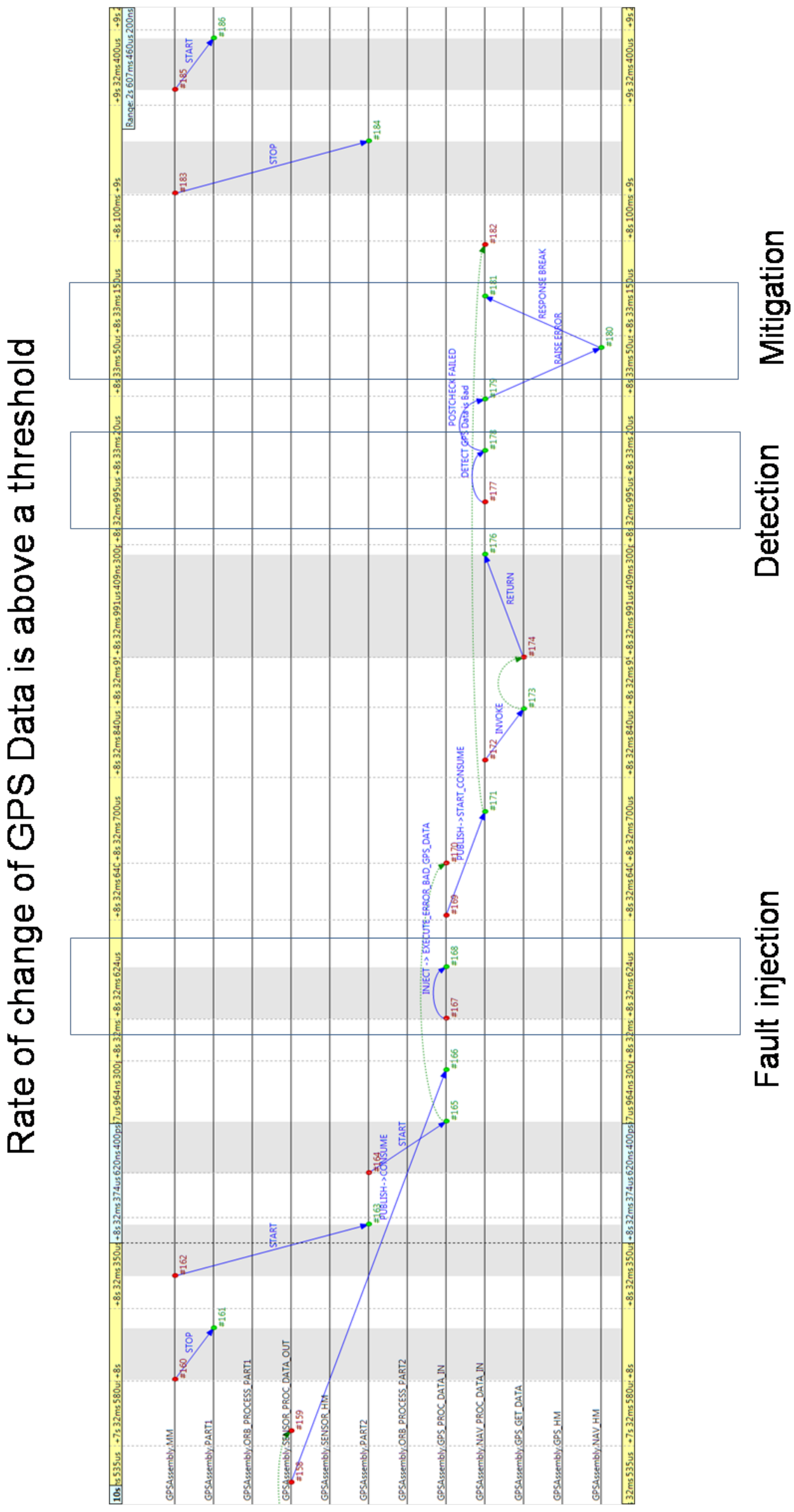


Fig. 17.   Sequence of Events for a BadGPSData case

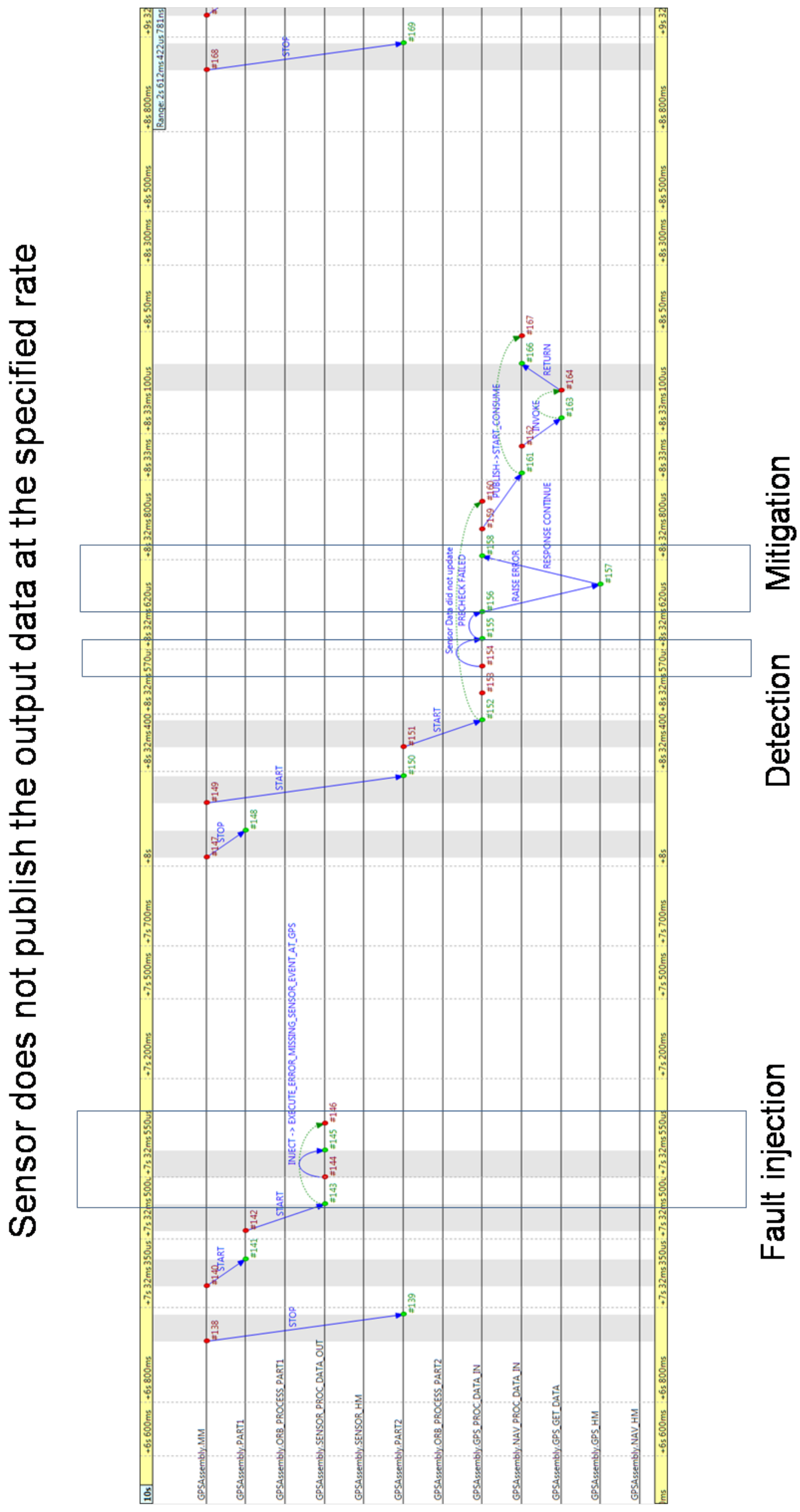

Sensor does not publish the output data at the specified rate
Fault injection
Detection
Mitigation
GPSAssembly.MM
GPSAssembly.PART1
GPSAssembly.ORB_PROCESS_PART1
GPSAssembly.SENSOR_PROC_DATA_OUT
GPSAssembly.SENSOR_HM
GPSAssembly.PART2
GPSAssembly.ORB_PROCESS_PART2
GPSAssembly.GPS_PROC_DATA_IN
GPSAssembly.NAV_PROC_DATA_IN
GPSAssembly.GPS_GET_DATA
GPSAssembly.GPS_HM
GPSAssembly.NAV_HM
STOP
START
INJECT -> EXECUTE_ERROR_MISSING_SENSOR_EVENT_AT_GPS
Sensor Data did not update
PRECHECK FAILED
RAISE ERROR
RESPONSE CONTINUE
PUBLISH->START_CONSUME
INVOKE
RETURN
Range: 2s 612ms 422us 781ns

Fig. 18. Sequence of Events for a MissingSensorEvent case

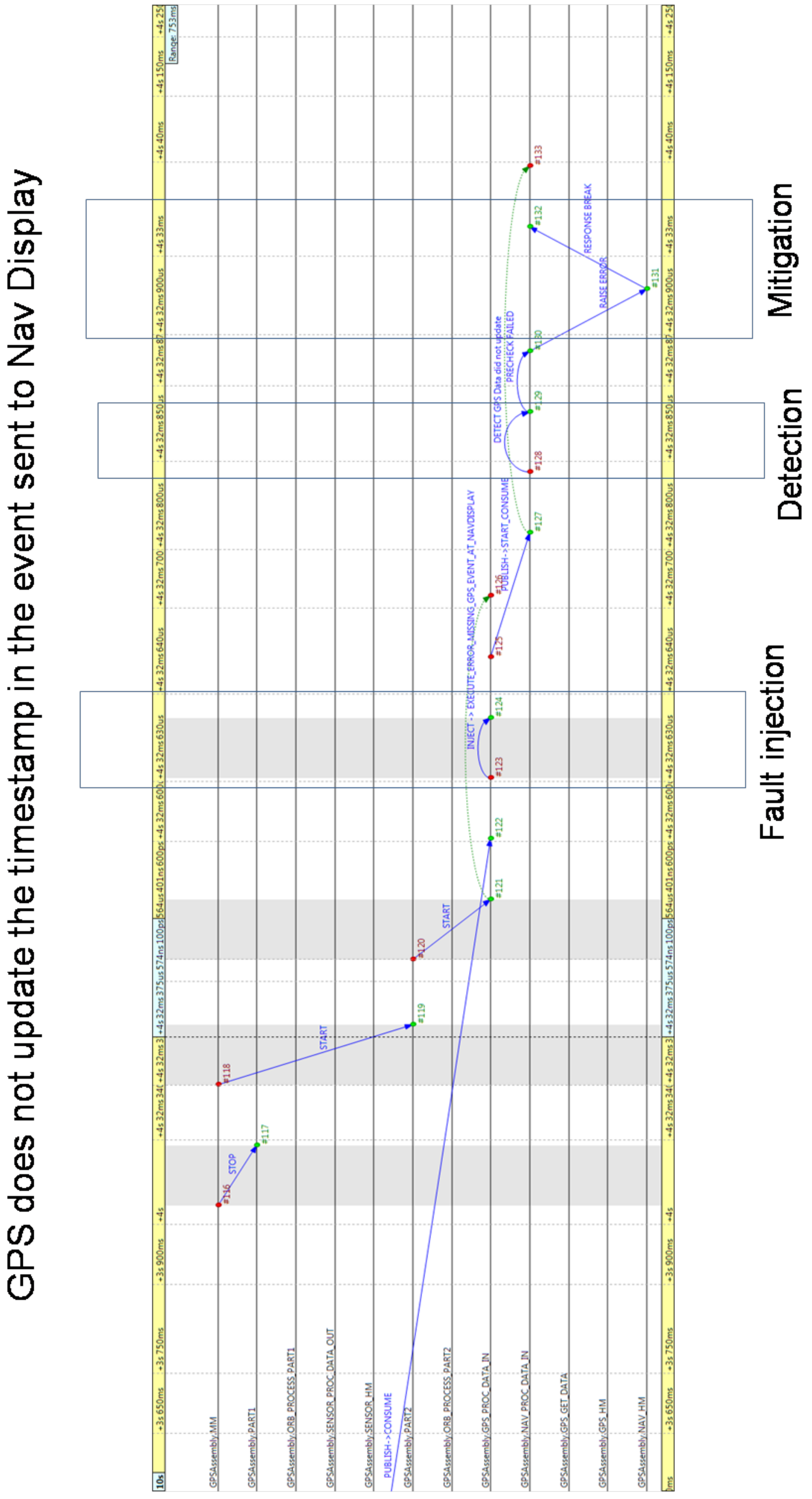

GPS does not update the timestamp in the event sent to Nav Display
Fault injection
Detection
Mitigation
GPSAssembly.MM
GPSAssembly.PART1
GPSAssembly.ORB_PROCESS_PART1
GPSAssembly.SENSOR_PROC_DATA_OUT
GPSAssembly.SENSOR_HM
GPSAssembly.PART2
GPSAssembly.ORB_PROCESS_PART2
GPSAssembly.GPS_PROC_DATA_IN
GPSAssembly.NAV_PROC_DATA_IN
GPSAssembly.GPS_GET_DATA
GPSAssembly.GPS_HM
GPSAssembly.NAV_HM
PUBLISH->CONSUME
STOP
START
START
INJECT -> EXECUTE_ERROR_MISSING_GPS_EVENT_AT_NAVDISPLAY
PUBLISH->START_CONSUME
DETECT GPS Data did not update
PRECHECK FAILED
RAISE ERROR
RESPONSE BREAK
Range: 753ms

Fig. 19. Sequence of Events for a MissingGPSEvent case