A Real-Time Component Framework: Experience with CCM and ARINC-653

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 while improving the productivity of developers. One possible approach is to use precisely defined software execution platforms that (1) enable the system to be composed from separate components, (2) restrict component interactions and prevent fault propagation, and (3) whose compositional properties are well-known. In this paper we describe the initial steps towards building a platform that combines component-based software construction with hard real-time operating system services. Specifically, the paper discusses how the CORBA Component Model (CCM) 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.

1 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 that ensure systems are dependable must keep up with this growth. One such technology is component-based software development: a concept that has been around since the early days of software engineering. Component-based development is built on the notion that software should be assembled from pre-fabricated and -tested software components that are customized and integrated via a component platform: a software framework. During the past two decades several software component models have been developed: COM and its follower .NET by Microsoft, the CORBA Component Model defined by OMG and implemented by many vendors, and Java Beans supported by Sun, just to name the three major ones. The component models define what a component is, how it can be customized, how it could be deployed on the platform, and how the components can interact via the platform. The primary goal of components is to promote reusability and to increase the productivity of developers. Furthermore, if the component model is well-designed, then the properties of the resulting system can be determined from properties of the components and how they are composed. Yet another potential benefit of using components is fault-management and -containment: the component framework can catch faults in components at run-time and take some appropriate action (e.g. restart the component) before the effect propagates to other components.

In spite of the apparent benefits of a component-based approach to development, little work has been done on applying these concepts to hard real-time systems. It is well-known that the complexity of hard real-time systems keeps increasing, and employing reusable components and robust composition techniques are crucial. In this paper, we describe the early results of a research effort that aims at developing a component framework for hard real-time systems. The software component framework provides capabilities similar to the ones in the CORBA Component Model (CCM), but it is built on the ARINC-653 platform abstractions. The bigger goal is to use this software component framework later as a foundation for software health management: an extension to classical software fault tolerance. We envision 'software health managers' that monitor components, detect anomalies, and take mitigation actions – all in a real-time context where dependability is required.

The paper is organized as follows. The second section provides the background for the real-time component framework we are constructing, while the subsequent sections detail our approach towards combining the 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.

2 Background

This section provides the necessary background on ARINC-653 platform services and our component model, two main technologies used in this work.

2.1 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 systems (RTOS) used in avionics. It has also been proposed as the standard operating system interface on space missions [7]. The APEX kernel in such systems is required to provide robust spatial and temporal partitioning to support functional separation between applications for fault-containment. The standard also describes a simple reactive health management service that supports recovery actions by using call-back functions that are mapped to specific error conditions in configuration tables at the partition/module/system level.

Spatial partitioning [11] ensures exclusive use of a memory region for a partition by ARINC processes (unless otherwise mentioned, a 'process' is meant to be understood as an 'ARINC Process' in the rest of this paper). An ARINC Process is similar to a thread in regular operating systems. Each partition has predetermined, statically allocated memory and its processes are prohibited from accessing memory outside of the partition's defined memory area. The memory protection is enforced by 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 isolate the low-criticality vehicle management components from safety-critical flight control components.

Temporal partitioning [11] refers to the strict temporal separation 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 is defined at configuration time. This provides deterministic scheduling whereby the partitions are allowed to access the processor or other hardware resources for only predetermined intervals of time. Temporal partitioning guarantees that a partition has exclusive access to the resources during its assigned time interval. It also guarantees that when the predetermined interval of execution time of a partition is over, the execution of the partition will be interrupted and the partition be placed 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 a resource is relinquished when the time slice for the corresponding partition expires.

2.2 ARINC Component Model

Our Component Model (Figure 1) is an extension of the standard CORBA Component Model (CCM). Like the CCM, component interactions can be either synchronous (serviced through provides/uses interfaces) or asynchronous (serviced through event sources/sinks). However, unlike the standard CCM where the functional logic belonging to an interface port is either executed on a new, dynamically created, or pre-existing but dynamically released worker-thread, here the functional logic for each port is executed as a separate ARINC process, which is defined and bound to the port during initialization. Due to the restrictions imposed by the ARINC specification, neither dynamic creation nor re-binding of the ARINC processes to a different port is permitted. Naturally, this indicates a shortcoming caused by constraints enforced by ARINC.

Furthermore, differently from CCM, each ARINC process bound to a specific port of the component is configured as either periodic or aperiodic. Periodic processes are released by the framework with a fixed frequency using a timer trigger. Aperiodic processes are released by (asynchronous) event-triggers or by (synchronous) method invocations. The possible combinations of synchronous and asynchronous interactions with periodic and aperiodic triggering gives rise to a rich set of potential component interaction behaviors. Another difference between this model and the standard CCM is the availability of interfaces for monitoring the component state and resource usage. This could be used by the software health manager (mentioned earlier) to detect anomalous behaviors in the component. Figure 2 shows a simple example assembly of components. The Sensor component contains an asynchronous publisher interface (source port) that is triggered periodically (every 4 sec). The event published by this interface is consumed by a periodically triggered asynchronous consumer/event sink port on the GPS component (every 4 sec). Note that the event sink process is periodically released, and each such invocation reads the last event published by the Sensor. If the Sensor does not update the event frequently enough, the GPS may read stale data. The consumer process in the GPS, in turn, produces an event that is published through the GPS's event publisher port. This event triggers the aperiodic consumer / event sink port on the Navigation Display component. Upon activation, the display component uses an interface provided by the GPS to retrieve the position data via a synchronous method invocation call into the GPS component.

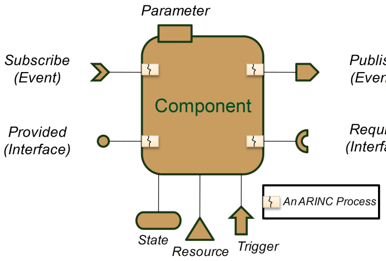


Figure 1. Component Model

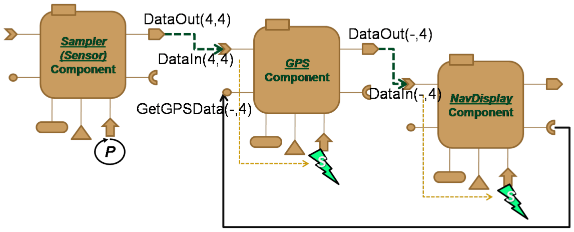


Figure 2. Example: Component Interactions. Here each interface is annotated with its (periodicity,deadline) in seconds.

3 Layering the Component Model on ARINC-653

ARINC-653 systems group Processes into spatially and temporally separated Partitions, with one or more Partitions assigned to each Module, and one or more Modules (processor hosts) form a System. The Partitions and their underlying Processes are created during system initialization and their dynamic creation is not supported. The user configures Partitions and their underlying Processes with their real-time properties (Priority, Periodicity, Duration, Worst Case Execution Time, Soft/Hard deadline etc.) The Partitions are precisely scheduled at run-time and their Processes are monitored to check for deadline violations. Processes within a Partition are independent, should share data only via the intra-partition services, 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 a single-item data message. Inter-partition communication is asynchronous and is provided using ports and channels that can be used for sampling or queuing of messages. Inter-process, intra-partition synchronization is supported through Semaphores and Events.

In a typical CCM deployment, the computational objects are grouped into Components that serve as run-time containers and form a layer between the objects and the underlying Object Request Broker (ORB). Each native operating system process contains an instance of the ORB that hosts the Components. Dynamic memory and resource allocation is permitted in a typical CCM system. If Components are configured as session-oriented, a new instance of the Component is created dynamically for each session request, if not then a single instance of the Component persists. The methods implemented inside a Component share and update 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 interfaces for synchronous method calls. 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.

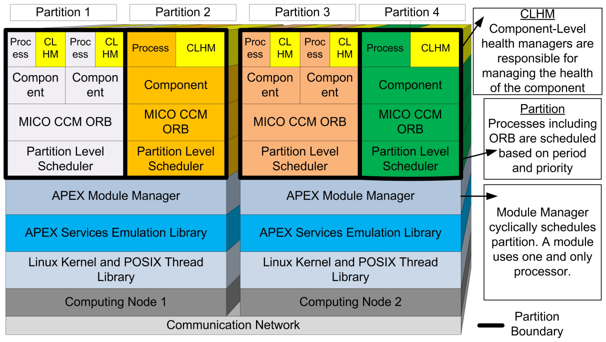

Partition 1
Partition 2
Partition 3
Partition 4
Proc ess
CL HM
Process
CLHM
Compon ent
Component
MICO CCM ORB
Partition Level Scheduler
APEX Module Manager
APEX Services Emulation Library
Linux Kernel and POSIX Thread Library
Computing Node 1
Computing Node 2
Communication Network
CLHM
Component-Level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

Figure 3. Layers of the ARINC Component Framework.

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. First we assume the existence of a host operating system that provides process and thread abstractions. An OS process is spatially separated from other processes and it can run multiple OS threads. There could be multiple hosts each running an OS, connected to a network. An OS process (that holds one ORB instance) is mapped to a ARINC-653 partition, and it can host multiple components. In CCM, interface methods of Components are executed in OS threads. In our implementation, each component interface method is an ARINC process that is mapped to an OS thread. All ARINC processes are instantiated with their respective interface method's periodicity (if periodic) and deadline. Note that ARINC-653 processes cannot be created during runtime; hence all processes executing interface methods are created at initialization time. The framework provides the necessary code to map an ORB request for executing a particular interface method to the releasing of the associated ARINC Process (i.e. OS thread). Table 1 summarizes the mapping between pure CCM concepts and how they are implemented in the combined framework.

Figure 3 describes the layers of our framework that implements the ARINC Component Model (ACM). The main

Table 1. Implementation of CCM concepts in the ACM framework. (RMI=Remote Method Invocation)

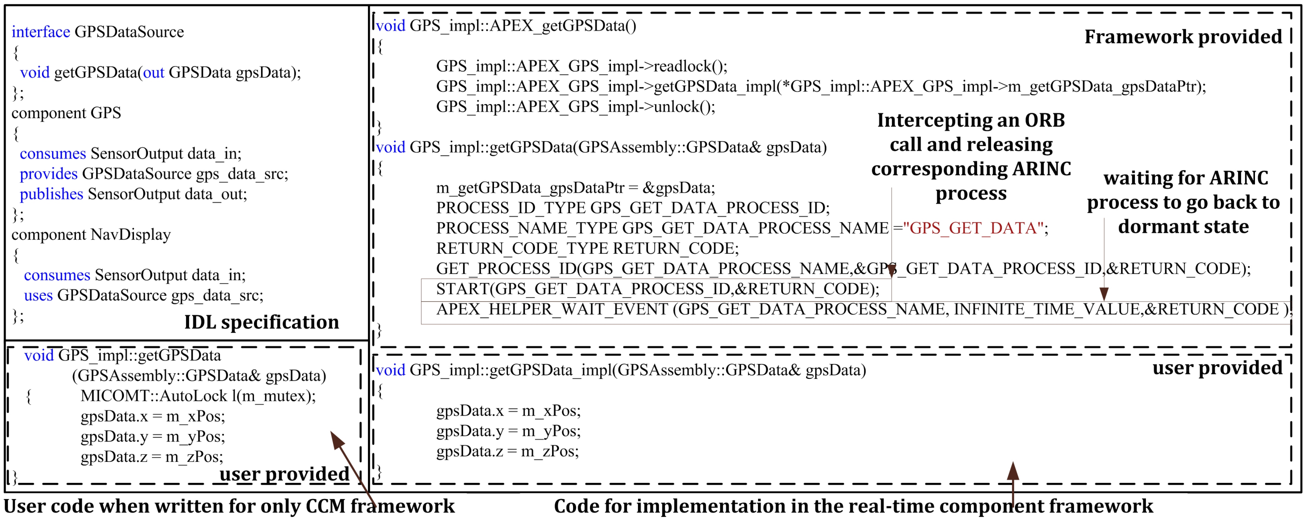
CCM	Target Properties		Features of ACM Implementation	APEX API Used
Host /Processor	N/A		An Apex module, mapped to a single CPU core.	Module
ORB Instance	N/A		An Apex partition, mapped to an OS Process.	Partition
Component Class	N/A		Data structure shared by related ARINC processes.	Semaphores
Component method	Periodic		Periodic process, mapped to an OS Thread	Process start, stop
Component method	Aperiodic		Aperiodic process, mapped to an OS Thread	Semaphores
Synchronous RMI	Periodic	Collocated	N/A	N/A
Synchronous RMI	Periodic	Non-collocated	N/A	N/A
Synchronous RMI	Aperiodic	Collocated	Caller method signals callee to release then waits for callee until completion.	Event, Blackboard
Synchronous RMI	Aperiodic	Non-collocated	Caller method sends RMI to release callee then waits for RMI to complete.	TCP/IP, Semaphore, Event
Asynchronous Publish-Subscribe	Periodic	Collocated	Callee is periodically triggered and polls event buffer (Blackboard) - validity flag indicates whether data is stale or fresh	Blackboard
Asynchronous Publish-Subscribe	Periodic	Non-collocated	Callee is periodically triggered and polls "Sampling Port" - validity flag indicates whether data is stale or fresh	Sampling port, Channel
Asynchronous Publish-Subscribe	Aperiodic	Collocated	Callee is released when event is available	Blackboard, Semaphore, Event
Asynchronous Publish-Subscribe	Aperiodic	Non-collocated	Caller notifies via TCP/IP, callee is released upon receipt	Blackboard, Semaphore, Event

purpose of this framework is to provide support for *developing* and *experimenting* with component-based systems using ARINC-653 abstractions on top of Linux. The secondary goal is to design the top layers: component and processes such that they can be easily rebuilt over an actual ARINC-653 kernel. The first two layers are a physical communication network and the physical computing platform. The next layer is the operating system. We have selected Linux as the **operating system** because it is widely available, supports a real-time scheduling policy (SCHED_FIFO), and provides an implementation of the POSIX thread library. *Memory partitioning* between Linux processes provided by the Linux Kernel is used to implement the spatial partitioning between ARINC-653 partitions.

APEX Services Emulation Library is the next layer. 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 messages and Blackboards enable processes to read, write, and clear single 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. Recall that, we implement ARINC-653 processes as POSIX threads. ARINC-653 processes, just like POSIX threads share the address space. 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. We have designed this layer such that it can be replaced by a real APEX kernel without affecting the layers on the top.

APEX Module Manager is the next layer. 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 the time duration of execution. It is known that potential partition jitter will occur if the periods associated with all partitions in a module are not harmonic i.e., between any given pair of partitions, the period of the first is an integer multiple of the second or vice versa [3, 9]. Moreover, the Process periods should be multiples of respective Partition periods to reduce process jitter.

The module manager is configured with a fixed cyclic schedule with pre-determined hyperperiod. The schedule is computed from the specified partition periods and durations. The module configuration 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 given scheduling window, which is specified with the offset from the start of the hyperperiod and a duration. The module manager is responsible for checking that the schedule is valid before the system can be initialized i.e. all scheduling windows within a hyperperiod can be executed without



```
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
Intercepting an ORB call and releasing corresponding ARINC process
waiting for ARINC process to go back to dormant state
user provided
User code when written for only CCM framework
Code for implementation in the real-time component framework

Figure 4. Equivalent implementation of a CORBA CCM interface in our framework.

overlap. Figure 5 shows the example execution time line of a module with two partitions and a hyperperiod of 2 seconds.

APEX Partition Scheduler, the next layer, is instantiated using 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. Upon *deadline violation*, the faulty process is prevented from further execution, which is the specified default action. It is possible to change this action to allow a restart.

Object Request Broker (ORB) is the next layer. This framework uses an open source CCM implementation, called MICO [17]. The main ORB thread is executed as an aperiodic ARINC-653 process within the respective partition. For controllability, the ORB runs at a lower priority than the partition scheduler. Since ARINC does not allow dynamic creation of processes at run-time, the ORB is configured to use a predefined number of worker threads (i.e. ARINC-653 Processes) that are created during initialization.

Component and Process Layers include the glue code (generated from the definitions of components their interfaces provided in an IDL file) and the user-provided implementation code. The developer is also responsible for specifying the necessary process properties such as periodicity, priority, stack size, and deadline. The framework provides glue code that maps each component interface method to an ARINC-653 process (see Table 1). 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 may engage read/write locks depending on whether they were specified as read-only or not.

The last layer on the top consists of **Component level Software Health Managers**. These are special ARINC processes that can take mitigation actions, if required. Details about this layer are discussed in [8].

4 Case study and Discussion

To test our framework, we developed the GPS example (Figure 2) on two ARINC partitions connected via a sampling port (for inter-partition communication). Table 2 shows all the ARINC-653 processes (and the partition they were deployed on) required to build this example. The developer had to write 148 lines of code to implement the example, while the 1027 lines of code were generated. Figure 4 shows a portion of the IDL. The bottom left hand side of the figure shows the code written by the user to implement the *getGPSData* interface for the GPS component, when written for pure MICO CCM implementation. The right hand side of the figure shows the equivalent code when written in the ACM 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.

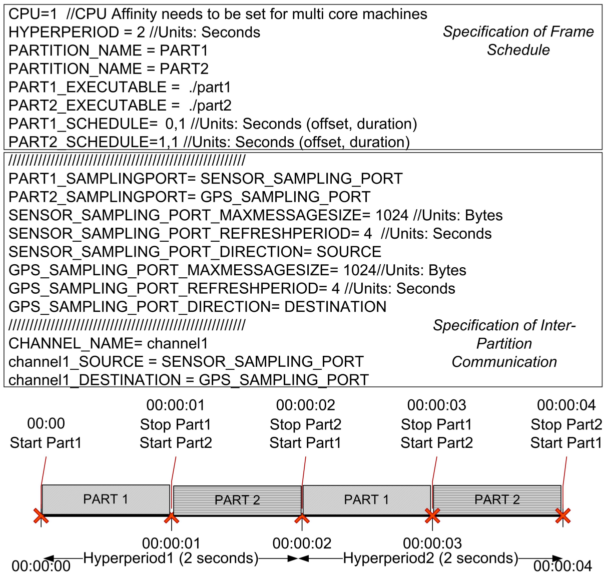
The configuration file for this experiment is shown in

Table 2. ARINC Processes created by the framework for the GPS example.

Part -tion	Process Name	Period	Dead -line	Type
Part 1	Part1 ORB Process	Aperiodic	Infinite	SOFT
Part 1	Sensor.DataOut	4sec	4sec	HARD
Part 1	Sensor HM	Aperiodic	Infinite	SOFT
Part 2	Part2 ORB Process	Aperiodic	Infinite	SOFT
Part 2	GPS.DataIn	4sec	4sec	HARD
Part 2	NavDisplay.DataIn	Aperiodic	4sec	HARD
Part 2	GPS.GetGPSData	Aperiodic	4sec	HARD
Part 2	GPS HM	Aperiodic	Infinite	SOFT
Part 2	Navigation HM	Aperiodic	Infinite	SOFT

Table 3. Summary of observed jitter.

Process	Std (μs)	Mean (μs)	Max (μs)
Part 1	2.26	68.29	72.47
Part 2	2.69	3.49	7.39
GPS.DataIn	0.96	228.16	229.12
Sensor.DataOut	0.74	153	153.77



```
CPU=1  //CPU Affinity needs to be set for multi core machines
HYPERPERIOD = 2 //Units: Seconds
PARTITION_NAME = PART1
PARTITION_NAME = PART2
PART1_EXECUTABLE =  ./part1
PART2_EXECUTABLE =  ./part2
PART1_SCHEDULE=  0,1 //Units: Seconds (offset, duration)
PART2_SCHEDULE=1,1 //Units: Seconds (offset, duration)
////////////////////////////////////////////////////
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
```

Figure 5. An example module configuration and the time line of events as they occur.

Figure 5. The hyper period was set to 2 seconds (period of both partitions was 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. Note that the period of periodic processes running inside the partition (Table 2) is 4 seconds, which is a multiple of the partition period. *This is required to prevent partition jitter as explained in the previous section.* 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 source port can be linked to more than one destinations). Table 3 contains the absolute jitter statistics for the two partitions and the two periodic processes inside the partition as measured from the start of the experiment running on Linux kernel 2.6.28 with high resolution timers. More experiments were performed to test the component level health managers (not included in this paper). Please refer to [8] for details on those experiments.

Lessons learned: **(1)** Conventional component frameworks rely heavily on dynamic threading, and they are typically not dealing with deadline violations. On the other hand, ARINC-653 relies on statically allocated Processes whose deadline violations are detectable. These two views are rather hard to reconcile, and our solution (one statically allocated ARINC Process per component method) is not optimal - it uses too many Processes. **(2)** 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. **(3)** A related problem is the use of dynamic memory allocation. The ARINC-653 specification requires that all runtime memory allocation 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. **(4)** We had mentioned in the previous section that access to a component's state variables is protected 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.

Extending CCM: (1) Our component model presented in Section 2 is an extension of the standard CORBA Component Model. We believe that a component level health management system will require interfaces for 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 one health monitor per partition. **(2)** The CORBA interceptors could be used to service the health monitors. 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 is to intercept interface specific requests and execute them in the respective component's health manager. **(3)** 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 must be terminated. However, all locks and resources used by that Process must be released (this is possible if locks are implemented using APEX semaphores) 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. **(4)** We also need 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.

Problems Identified: During our experiments we came across issues that are important to emphasize: **(1)** we discovered that in typical CCM implementations like MICO [17] and CIAO [5], the publish-subscribe 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 have implemented 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. Inter-partition CCM event-based communication is mapped to sampling and queuing ports. **(2)** 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. **(3)** ARINC-653 specification does not permit changing the properties of an ARINC process (e.g. WCET) at runtime. This decision is primarily governed by the analysis advantages that such static configuration provides. However, as a consequence, we have to create a new process for each functional logic that needs to run as a separate thread in a component. This approach results into a large number of processes for a bigger software assembly.

5 Related work

5.1 Real-time Software Component Frameworks

An approach to objects based on time-triggered (periodic) and event-triggered (aperiodic) methods has been presented in [12]. The approach described is implemented in the form of object structures, and many concepts are similar to our work. However, there are two differences: we rely on an industry standard specification, ARINC-653, as the underlying platform, and we build a framework on top of that to provide specific services for component interactions and scheduling.

TinyOS [21], ControlShell [18], eCos [10] , Koala [22] are component-based frameworks geared towards resource constrained embedded devices. They are primarily event-triggered and rely on design-time checks and tests to ensure correctness of implementation. They do not focus on spatial and temporal partitioning.

The GENESYS (GENeric Embedded SYStem) [16] research project has developed a cross-domain component-based architecture for embedded systems. It has been designed for achieving (1) *compositionality* to allow system designers to compose systems using independently developed and tested components (unit of abstraction), (2) *robustness* to provide fault containment and selective restart of components upon failure, and (3) *energy efficiency* by integrating resource management in the platform design. An important principle followed in GENESYS architecture is the strict separation of computational components and communication paradigm. This makes it possible to design and analyze the two systems in separation. The work presented in this paper uses ARINC-653 as the underlying platform which provides the temporal and spatial separation between applications. However, it does not restrict the behavior of the underlying communication protocol. In future, we will investigate the use of the Avionics Full-Duplex Ethernet (AFDX), which is a time-deterministic network defined in ARINC-664 standard [1].

CIAO [5] and PRISM [20] are two component models built upon the real-time CORBA implementations. PRISM employs a static component allocation and configuration policy and supports publish/subscribe paradigm. CIAO supports both dynamic and static component configurations. Both CIAO and PRISM have been designed for minimum overhead and priority preemptive systems. However, the IDL specification and generated code does not specify deadlines. Deadline violation monitoring is left to application level user supplied code. Our work presented in this paper discusses the enhancements and restrictions required to MICO or CIAO so that it can be ported to a hard real-time operating system that supports temporal and spatial partitioning.

Kuz et al. presented a component model called CAmkES in [13]. 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 x86 architecture as it allows more flexibility in experimenting with this technology.

Delange et al. recently published their work on POK (PolyORB Kernel) [6]. It uses a modeling framework based on the ARINC 653 Annex for AADL, a modeling language for safety-critical systems, to automatically configure and deploy processes and partitions upon POK, which is ARINC-653 compliant. However, this work does not implement a component framework over ARINC-653. DIANA [19] is a new project for implementing an avionics platform called Architecture for Independent Distributed Avionics (AIDA) using Java as the core technology. However, the choice of using JAVA as the core technology increases the runtime complexity. Using Java threading model requires the system to add another layer of scheduling above the operating system, which makes the analysis of software assembly very difficult. Another issue with using Java is the complexity in estimating and bounding memory usage per thread, which is a critical requirement in the ARINC-653 standard.

Lakshmanan and Rajkumar presented a distributed resource kernel framework used to deploy real-time applications with timing deadlines and resource isolation in [14]. 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.

5.2 Schedulability analysis for ARINC-653 systems

Schedulability analysis is important for a hard-real time system. Audsley et al. presented a discussion on the ARINC-653 standard and schedulability analysis for such systems in [3]. They showed that partitions can be analyzed in isolation by aggregating the timing requirements of all other partitions. Work on a similar problem was recently reported by Easwaran et al. [9]. They focused on using compositional analysis techniques and took into account the process communication, jitter, and preemption overheads. 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 [15]. 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.

Burns and Lin [4] 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 [2]. Consequently, they can only validate scheduling for non-preemptive systems with known computation time for all tasks. Therefore, this method can be used only for analyzing ARINC-653 partitions which are statically scheduled and cannot be used for analyzing ARINC-653 processes, which can be suspended during execution by other processes.

All the algorithms mentioned above require the knowledge of the worst case execution time (WCET) associated with each task. However, estimating WCET is difficult and as a consequence it is possible that deadlines are violated during run-time. Therefore, our framework actively monitors all deadline. Any violation results in the default action of stopping the faulty process. An API is available to restart the faulty Process after a reset.

6 Conclusions

This paper presented our first steps towards building a real-time component model and the underlying framework. 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 recognize that further developments are needed that integrate components with a hard real-time platform.

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] R. L. Alena, J. P. Ossenfort, K. I. Laws, A. Goforth, and F. Figueroa. Communications for integrated modular avionics. In *Proc. IEEE Aerospace Conference*, pages 1–18, 3–10 March 2007.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] N. Audsley and A. Wellings. Analysing APEX applications. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 39, 1996.

[4] A. P. Burns and T. M. Lin. An engineering process for the verification of real-time systems. *Formal Aspects of Computing*, 19(1):111–136, March 2007.

[5] CIAO. http://download.dre.vanderbilt.edu/.

[6] 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*, pages 30–43.

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

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

[9] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal. A compositional framework for avionics (ARINC-653) systems. Technical Report MS-CIS-09-04, University of Pennsylvania, Feb 2009. http://repository.upenn.edu/cis_reports/898/.

[10] eCos. http://ecos.sourceware.org/.

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

[12] K. Kim. Object structures for real-time systems and simulators. *Computer*, 30(8):62–70, Aug 1997.

[13] I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699, 2007.

[14] 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*, 0:195–204, 2008.

[15] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, 2005.

[16] R. Obermaisser and H. Kopetz, editors. *Genesys An Artemis Cross-Domain Reference Architecture For Embedded Systems*. Sudwestdeutscher Verlag fur Hochschulschriften AG, 2009.

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

[18] S. Schneider, V. Chen, J. Steele, and G. Pardo-Castellote. The controlshell component-based real-time programming system, and its application to the marsokhod martian rover. *SIGPLAN Not.*, 30(11):146–155, 1995.

[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*, pages 169–178, 2009.

[20] M. Schulte. Model-based integration of reusable component-based avionics systems - a case study. In *ISORC 2005*, pages 62–71.

[21] Tinyos. http://webs.cs.berkeley.edu/tos/.

[22] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.