A Component Model for Hard Real Time Systems: CCM with ARINC-653

Abhishek Dubey Gabor Karsai Nagabhushan Mahadevan
Institute for Software Integrated Systems, Vanderbilt University,
Nashville, TN 37212, USA

Abstract

Size and complexity of software in safety critical system is increasing at a rapid pace. One technology that can be used to mitigate this complexity is component-based software development. However, in spite of the apparent benefits of a component-based approach to development, little work has been done in applying these concepts to hard real time systems. This paper improves the state of the art by making three contributions: (1) we present a component model for hard real time systems and define the semantics of different types of component interactions; (2) we present an implementation of a middleware that supports this component model. This middleware combines an open source CORBA Component Model (CCM) implementation (MICO) with ARINC-653: a state of the art RTOS standard, (3) finally; we describe a modeling environment that enables design, analysis, and deployment of component assemblies. We conclude with a discussion of lessons learned during this exercise. Our experiences point towards extending both the CCM as well as revising the ARINC-653.

1 Introduction

Core functions and system integration effort in complex cyber-physical systems, such as aircraft are increasingly becoming reliant on software. This has led to an exponential increase in size and complexity of software associated with these systems [32, 6]: avionics architectures have evolved from independent analog avionics in 40's, to federated avionics of 60's, to the integrated avionics of 80's, and finally to the advanced integrated avionics of post 2000 era. One way to address the rise in system complexity is to use component-based software along with robust composition techniques for constructing these systems; a concept that has been around since the early days of software engineering.

Component-based software development is based on the notion that software should be assembled from pre-fabricated and pre-tested components, which encapsulate parts of a software system that implement a specific service or a set of services. Several software component models have been developed in the past two decades. These include COM and .NET by Microsoft, the CORBA Component Model defined by OMG and implemented by many vendors, and Enterprise Java Beans from Sun/Oracle, just to name the three major ones. The component models define what a component is, how it can be customized, how it can be deployed on the platform, and how the components

can interact via the platform. The primary goal of this approach 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 interacting components [35, 42]. Yet another potential benefit of using components is fault-management and -containment: the component middleware 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.

There has been considerable interest in developing a component middleware for soft real time systems [10, 39, 12]. Most implementations have relied on different implementations of RT-CORBA [21]. However, in spite of the apparent benefits of a component-based approach to development, little work has been done in applying these concepts to hard real time systems. It is known that the complexity of such hard real time systems keeps increasing over time, and using reusable components along with robust composition techniques is crucial.

This paper is an extension of our work published earlier in [17]. The contributions of our work described here are:

1. We present a component model that is suitable for hard real time systems. The guiding principles of our design are static memory allocation for determinism, spatial and temporal isolation between components of different criticality, specification and adherence to real time properties such as periodicity and deadlines, and providing well-defined compositional semantics. Instead of starting afresh we chose to develop our work based on the existing OMG CCM standard[1] because it is one of the widely used component model and software developers are already familiar with it.

2. Next, we describe the design and implementation of the middleware that supports the component model. This middleware is built by extending an open source CORBA Component Model (CCM) implementation called MICO [34] and layering it on top of an emulation of ARINC-653 platform services [1]: the state of the art standard in Integrated Modular Avionics architecture [44]. We had to use an emulated implementation because of lack of access to an actual ARINC-653 implementation.

3. Finally, we describe the domain specific modeling environment for this framework, constructed using Model Integrated Computing tools available at [4]. It allows the application developers to model components and the set of services they provide (without detailing the implementation and independently of the actual deployment configuration); and capture real time properties and resource requirements. This allows system integrators to configure software assemblies using precisely defined compositional semantics, specify the deployment topology, and perform design-time checks. These tools and the runtime are available for at `https://wiki.isis.vanderbilt.edu/mbshm/index.php/ACMTOOLSUITE`.

The paper is organized as follows. Section 2 provides the background on the ARINC-653 standard and the CORBA Component Model. Section 3 describes our extensions to the CORBA component model in detail. Sections 4, 5 and 6 detail our approach towards combining the CORBA Component Model with the hard real time ARINC-653 platform services and present our results. We conclude with related works and a summary in sections 7 and 8.

[1] http://www.omg.org/spec/CCM/4.0/

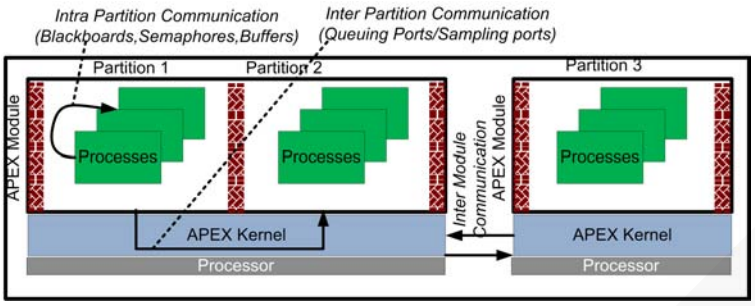

Figure 1: ARINC 653 architecture.

2 Background

This section provides the necessary background on ARINC-653 platform services, the CORBA Component Model and Model-Integrated Computing, three main technologies used in our work.

2.1 ARINC-653 Platform Specification

The ARINC-653 software specification describes the standard Application Executive (APEX) kernel and associated services that are 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 [15].

ARINC-653 systems (see figure 1) group *processes* into spatially and temporally separated *partitions*, with one or more partitions assigned to each *module* (i.e. a processor), and one or more modules forming a *system*. **Spatial partitioning** [23] ensures exclusive use of a memory region by an ARINC partition. It also guarantees that a faulty process in a partition cannot ruin the data structures of other processes in other partitions. Such space partitioning can, for example, be used to isolate the low-criticality vehicle management components from safety-critical flight control components. Each partition has predetermined statically allocated memory. The ARINC processes hosted within a partition are prohibited from accessing memory outside of the partition's defined memory area. This memory protection is enforced by memory management hardware.

Temporal partitioning [23] ensures exclusive use of the processing resources by a partition. A fixed periodic schedule is used by the RTOS to share the resources between partitions. This deterministic scheduling ensures that each partition is allowed exclusive access to the processor or other hardware resources within its predetermined execution interval. It also guarantees that when the predetermined execution interval of a partition is over, the partition's execution will be interrupted, the partition will be placed into a dormant state and the next partition in the schedule order will be granted exclusive access to the computing resource, i.e. the processor. 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.

This architecture provides functional separation between applications for fault-containment. The partitions and their underlying processes are created during system initialization. Dynamic creation of processes while the system is running is not supported. The user configures partitions and their underlying processes with their real time properties (priority, periodicity, duration,

Table 1: Communication mechanisms available in an ARINC-653 compliant operating system.

Communication Type	Destructive Read	Messages Buffered	Co-located (in the same Partition)
Blackboard	No	No	Yes
Buffer	Yes	Yes	No
Sampling Port	No	No	Yes
Queuing Port	Yes	Yes	No

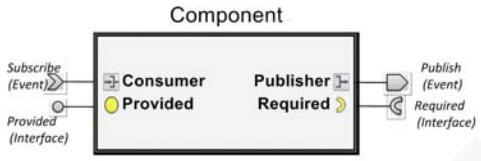

Figure 2: CORBA Component Model

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 share data via the intra-partition services. Intra-partition communication is supported 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 provided using ports and channels that can be used for *sampling* or queuing of messages. Synchronization support for processes within a partition is provided through semaphores and events. Table 1 summarizes these communication mechanisms.

2.2 CORBA Component Model

The CORBA Component Model (CCM) [43], illustrated in Figure 2, is a standard defined by the Object Management Group (OMG). It extends the CORBA object model and defines features that allow software developers to implement, manage and deploy software units (components) that integrate commonly used functionality and allow for greater software reuse.

The component interfaces are described (using an interface definition language) in terms of attributes and ports exposed by the component. While the attributes are data entries, ports correspond to the interaction points with other components. Ports could belong to any of the following four types:

(a) Facets: Facets or provided interfaces are sets of services (methods) that a component exposes to other components. Each service can be called from another component synchronously via a CORBA two-way operation or asynchronously via CORBA's asynchronous method invocation. A service can be also invoked using the CORBA one-way operation.

(b) Receptacles: Receptacles or required interfaces are sets of services that a component needs from another component to perform its task. The client component is required to connect its receptacle to a compatible facet of before it can issue the service calls.

(c) Event sources/sinks: Components can publish or subscribe to different types of events using these ports. An event is a simple, flat data structure. Typically, an event service daemon or broker is responsible for logically connecting a source port (publisher) in one component to sinks (consumers) in other components. This event service is also responsible for routing the messages

from the sources to sink.

(d)Attributes (with setters/getters): The values of CORBA component attributes can be set and queried by external entities; the attributes may represent the state of the component.

In a typical CCM deployment, the computational objects are grouped into Components hosted inside a CCM Container, which is associated with 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. The alternative is to configure components as **service-oriented**, which implies that a single instance of the Component persists across all invocations. All interactions between Components happen through ports that are used to publish or receive subscribed events, or ports that provide or use interfaces for method calls. Any incoming request is launched and serviced in worker threads dynamically obtained from a thread pool by the middleware. The communication between the objects is achieved through the services provided by the Component middleware layer and the underlying ORB. Additional synchronization support from the ORB service libraries and the underlying OS is also available.

2.3 Model-Integrated Computing

Model Integrated Computing (MIC) [3] is an approach to the development of complex software systems. The key idea is to use models in all phases of the development (analysis, design, implementation, testing, maintenance and evolution). The Model-driven Architecture (MDA) [2] of the Object Management Group (OMG) is a parallel approach with overlapping directions. MIC incorporates the creation of domain-specific, model-based abstractions, which serve to capture relevant aspects of a target system. Such abstractions are available in the form of a domain-specific modeling language which is used by developers to create multi-aspect models of the system. These models can then be programmatically traversed and transformed to produce (or modify) code, other engineering artifacts, etc. Often, models are transformed into alternate but equivalent representations, which can be used by external analysis and simulation tools to verify certain properties of the system.

In the MDA approach, the key notion is the use of Platform-Independent Models (PIMs) to describe the system in high-level terms, then refine these models (possibly using model transformations) into Platform-Specific Models (PSMs) which are then directly used in the implementation (which itself could - wholly or partially - be generated from models). In the MIC approach, the use of Domain-Specific Modeling Languages (DSMLs) is advocated. A domain-specific modeling language (DSML) allows a designer to describe objects in terms of the domain-specific abstractions rather than in terms of traditional computer languages. The Generic Modeling Environment (GME) [29] is a freely available tool, which provides a platform for Model Integrated Computing design and development.

Next section describes our design of a component model for hard real time systems. We pay special attention to the discussion, which highlights the enhancement in this component model with respect to CCM.

3 A Component Model for Hard Real Time Systems

Any component model that aims at being suitable for hard real time systems shall address the following issues.

1. It must support the specification of real time properties such as periodicity and deadline. CCM standard natively does not support such specifications.

2. It must be analyzable. That is, we must be able to analyze the properties of the system from the underlying component assembly. For this, the component model must have well-defined interaction semantics.

Moreover, given that typical hard real time systems require determinism, components must support static memory allocation; i.e. the resources required for execution of tasks must be reserved at system initialization. An advantage of this approach is that it avoids memory fragmentation, which, if present, requires the RTOS to spend precious computing cycles managing the memory. The consequence of static memory allocation is that components can only support service life cycle i.e. a session-based component cannot be created dynamically based on an incoming request. This is especially true in hard real time systems, where all possible tasks must be known beforehand to ensure that they can be scheduled.

Another feature, which is not a requirement for hard real time systems but is important to have, is spatial and temporal partitioning. As discussed in section 2.1, a safety-critical real time system can group processes into separate partitions with different criticality levels and still allow them to execute in the same core module, without affecting one another spatially or temporally. This is a good feature to have in a software assembly, where components with different criticality levels can be isolated from each other, which in turn provides fault isolation.

Remark 1 *We have named our component model ARINC-653 Component Model (ACM) because ARINC-653 standard is the current state of the art for safety critical real time systems that supports the two platform principles mentioned in previous paragraphs. However, we should clarify that this component model is not necessarily tied to ARINC-653 and can be implemented on any other platform that provides similar services.*

Figure 3 illustrates the internals of our component model. A component can have four different kinds of ports: consumer port, publisher port, provided interface port (similar to a facet in CCM) and required interface port (similar to a CCM receptacle). A publisher port is a source of events: this port is used to produce events that will be consumed by another component/s. A publisher port needs to be triggered to publish an event (probably read from some internal state variable or a hardware source). This triggering can be either periodic or aperiodic (sporadic). While, a periodic publisher is triggered at regular intervals by a clock to supply data, an aperiodic publisher is invoked (sporadically) by an internal method, possibly the implementation code of another port. A consumer port, as the name suggests, acts as a sink for events. Like a publisher port, it can be triggered periodically (by a clock) or aperiodically (by the arrival of an event) to consume an event. An aperiodic consumer consumes all the events published by its publisher on a FIFO basis (destructive read) and a periodic consumer samples the events published at a specified rate (nondestructive read).

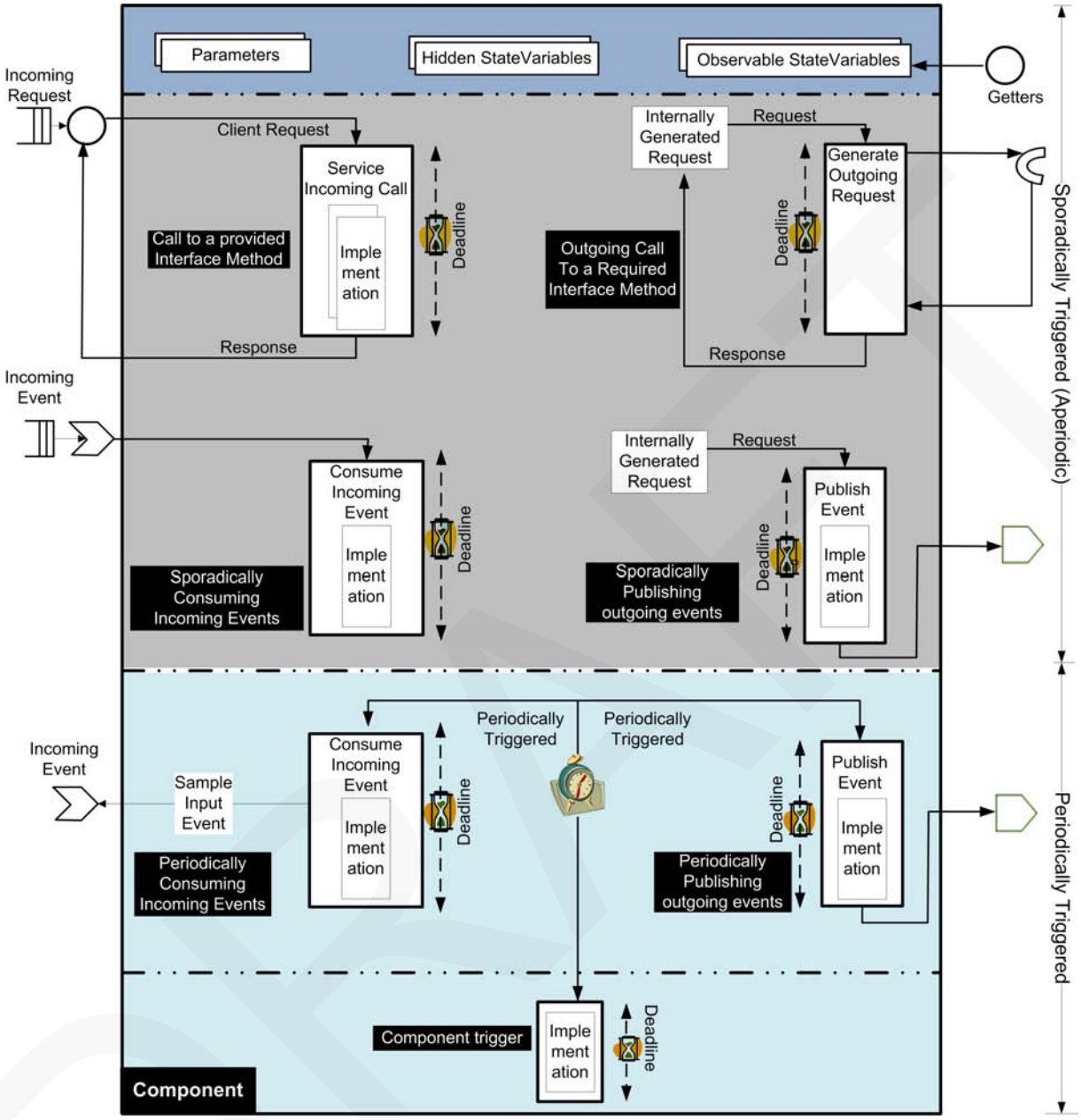

Figure 3: ARINC Component Model (ACM)

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

Two new concepts exist in our extension to the CCM: *state variables*, which are similar to attributes in CCM but cannot be modified from outside component, and *component triggers*, which are internal periodically activated methods within a component that can be used for internal bookkeeping and checking state invariants.

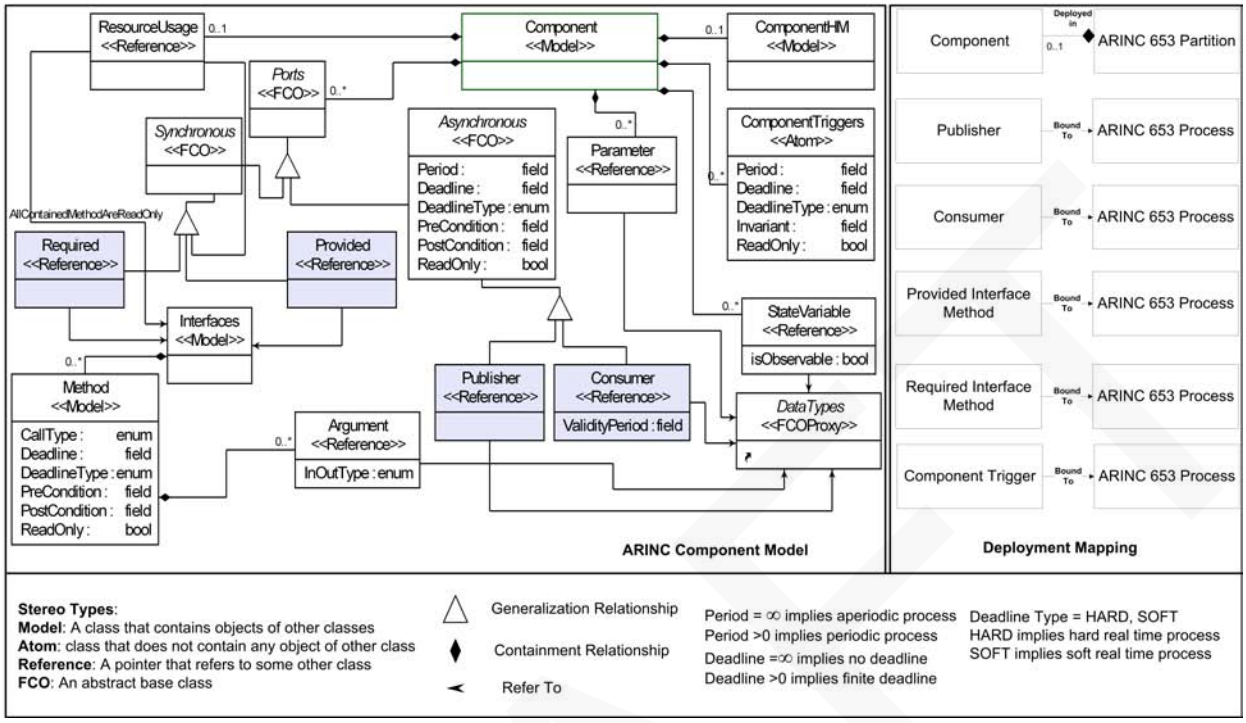

Figure 4: ACM Metamodel. The colored boxes indicate the four types of ports, which can be used to interact with other components.

All the ports - publisher, consumer, facet, and receptacle - and the component trigger method have to finish their unit of work within a specified deadline. This deadline can be qualified as HARD (strict) or SOFT (relatively lenient). A HARD deadline violation is an error that requires intervention from the underlying middleware. A SOFT deadline violation results in a warning. The soft deadline warning is sent to the component health manager- discussed in section 3.1, which can act upon it if configured to do so.

Like the deadline, all implementations can specify another property that must be respected: *contracts*. These contracts are expressed as pre-conditions and/or post-conditions, and any contract violation results in an error. This concept is based upon the logic system identified by Hoare [24]. These are discussed further in section 3.2. Next section describes the metamodel i.e. the relationship and attributes associated with these ports.

3.1 ACM Metamodel

Figure 4 describes the metamodel of our extensions to the CORBA Component Model. This metamodel has been designed in a UML based modeling language called MetaGME [29, 4]. It shows the relationship between all the ports and different attributes that can be set. The main concepts are as follows:

(a) State Variables: As described in the previous section, these represent the externally observable state of the component. Unlike CCM attributes they cannot be modified from outside.

(b) Parameters: are configuration attributes which once set during initialization remain constant during the life cycle of that component.

(c) Asynchronous Ports: These kinds of ports participate in asynchronous interactions. There are two kinds of asynchronous ports, **Publisher** and **Consumer**. We discussed both of them in the previous section. The data type of an event is specified by the type entity referred to by the port. Both ports can be further qualified by attributes such as period, deadline, pre-conditions and post-conditions, see Table 2. If the period of a publisher is set to infinity, it is called aperiodic. Otherwise, it is periodic. Periodic publishers are typically activated by the middleware, automatically, while aperiodic publishers are activated by another method within the component. Periodic consumers are automatically and periodically activated by the middleware, aperiodic consumers are activated when they have events to consume.

(d) Synchronous Ports: These ports are used for either requesting some service from another component or servicing an incoming request. There are two main kinds of synchronous ports, **Provided** and **Required**. Each provided port contains implementation of a collection of **methods**. These methods can be further qualified by filling in their attributes e.g. call type, deadline, pre-conditions and post-conditions. (see Table 2). Furthermore, each method contains a number of arguments. These arguments can be marked to be used strictly for sending input (IN) to the facet, strictly for sending output (OUT) to the receptacle, or to be used for both input as well as output (INOUT). The third kind, **Resource Usage Monitoring Interface** is used for monitoring the component resource usage. By design, this interface can only contain read only methods i.e. they cannot change the internal state of the component.

(e) Component Triggers: A component can contain a number of internal methods, known as component triggers. These methods are periodically triggered. They must have a finite, non-zero period. Typically, they are used for record keeping and invariant checking of the component. These methods are further qualified by filling in attributes such as period, deadline and invariant (see Table 2).

(f) Component Health Managers: Component-level health managers (CLHM) for software components detect anomalies, identify and isolate the fault causes of those anomalies (if feasible), prognosticate future faults, and mitigate effects of faults – on the level of individual components. CLHM is implemented as a 'side-by-side' object that is attached to a specific component and acts as its health manager. It provides a localized and limited functionality for managing the health of one component, but it also reports to higher-level health manager(s): the system health manager - all in a real time context where dependability is required. A detailed discussion on this topic is out of scope for this paper. Interested readers are referred to [18, 16].

Deployment: Unlike the standard CCM where the functional logic belonging to an interface port is executed on a new, dynamically created, or pre-existing but dynamically released worker-thread, in ACM the functional logic for each port is executed on a statically allocated schedulable unit. This choice is guided by our first design principle of static memory allocation, which restricts dynamic creation. On an ARINC-653 operating system, these schedulable units are ARINC-653 processes. Furthermore, each component is deployed on one ARINC-653 partition. Multiple components can reside in the same partition.

3.2 Correctness Contracts

Each functional entity in ACM can be annotated with correctness criteria, which are specified as pre-conditions and post-conditions. We envision that these conditions should be specified over the current value, or the history of the value, or rate of change of values of certain data elements.

Table 2: Attributes of Component Entities

Name	Type	Values	Default	Remark	Applicable To
Validity Period	double	$[0, \infty)$	∞	Marks the time-limit for an event to be valid (not stale).	Consumers
ReadOnly	Boolean	True, False	∞	Marks if the port can update the state of the component or not	All ports and component triggers
Deadline	double	$[0, \infty)$	∞	Marks the time-limit by which the process should finish execution.	All ports and component triggers
Deadline Type	ENUM	HARD SOFT	HARD	Marks the nature of the deadline.	All ports and component triggers
Period	double	$[0, \infty]$ for SOFT deadline. $[Deadline, \infty]$ for HARD deadline	∞	Marks the rate at which the scheduler launches the process. Notice that the maximum response for hard deadline tasks should be less than or equal to the period.	All ports and component triggers. Synchronous ports are always aperiodic.
Call Type	ENUM	TWO WAY ONE WAY	TWO WAY	Marks whether an RMI call is blocking or non-blocking(same as in CCM/CORBA)	Methods on Synchronous ports
Post-Condition	string	N/A	N/A	The contract to be satisfied at the end of the execution.	All ports
Pre-Condition	string	N/A	N/A	The contract to be satisfied at the beginning of the execution.	All ports
Invariant	string			The contract to be satisfied during the operation of trigger.	Component triggers

These data elements can be part of the (a) the event-data of asynchronous calls, or (b) the function-parameters of synchronous calls, or (c) the state variables of the component, or (d) resource usage of the component. While pre-conditions are assumptions that must be true before execution of a call, post-conditions are guarantees, which will be true after the execution. This type of reasoning is critical in achieving modular certification of software components [35].

Remark 2 *Given that multiple components can be deployed in a partition, it is possible that one component can modify the state variables of the other component. Therefore, it is important to ensure that components are coded in a way that this does not happen. One way to address this is to use our model-based process, described later in this paper. This process, if followed, ensures that the components even if located in the same partition exchange information only though their external interfaces and do not address each other's memory directly.*

3.3 Semantics of Component Interactions

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

3.3.1 Asynchronous Interactions

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

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

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

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

3.3.2 Synchronous Interactions

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

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

The restriction on synchronous interactions has been relaxed to allow CORBA style one-way calls. When such methods are invoked, the requires port performs a non-blocking call. It returns without waiting for the provides port to finish its operation. There are no return values in such calls. However, one should note that even though the call is made in a non-blocking fashion it is different from an asynchronous interaction. While, a publisher does not fail if a consumer fails to

consume the message properly, a one-way call via the middleware will result in an exception if the target provided port is not available.

Remark 3 *Safe component interactions must satisfy the following constraints: (a) Deadline of a requires port must be greater than or equal to the deadline of the interacting provided port. (b) Validity period of a consumer must be greater than the periodicity of the publisher. Otherwise, a consumer can possibly receive data that it considers stale. (c) The contract imposed by the post-condition of a component providing a service or publishing an event must be stricter than the pre-condition checked by the interacting destination component. If this is not true, the source component might send data, which is locally valid but will violate pre-condition of the destination component.*

Remark 4 *It is important to point out that it is possible to identify specific faults and their propagation pattern based on component interactions - periodic publisher/periodic consumer, periodic publisher/aperiodic consumer aperiodic publisher/periodic consumer, aperiodic publisher/aperiodic consumer, and synchronous interactions. While the interaction ports can be customized (by the event-data-types published/consumed, interfaces/methods exposed, periodicity, deadline etc.), their fundamental behaviors and interaction patterns are well defined. Additionally, capturing the details of how the data and control flows within a component by modeling the data dependencies between its ports and its state variable and modeling the control dependencies between different ports further assists in capturing the fault propagation within the component. This approach is similar to the failure propagation and transformation calculus described by Wallace [42].*

Example 1 *Figure 5 shows a simple example assembly of components. The Sensor component contains an asynchronous publisher interface (source port) that is triggered periodically (every* 4 *msec). The event published by this interface is consumed by a periodically triggered asynchronous consumer/event sink port on the GPS component (every* 4 *msec). 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.*

Next, we describe the implementation of a framework that provides services necessary for constructing software applications from ARINC-653 components.

4 Implementation of a Framework for Enabling ARINC-653 Component Model

Given that our component model described in the previous section is an extension of CCM, we took the approach of using and extending an existing implementation of CCM and layering it on top of a library implementing ARINC-653 platform abstractions. During the exercise, we discovered that there are certain incompatibilities between the two. These are discussed later in section 6.

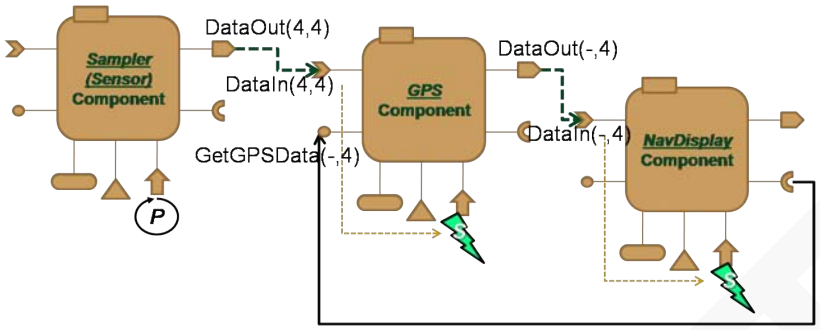

Figure 5: Example: Component Interactions. Here each interface is annotated with its (periodicity, deadline) in seconds.

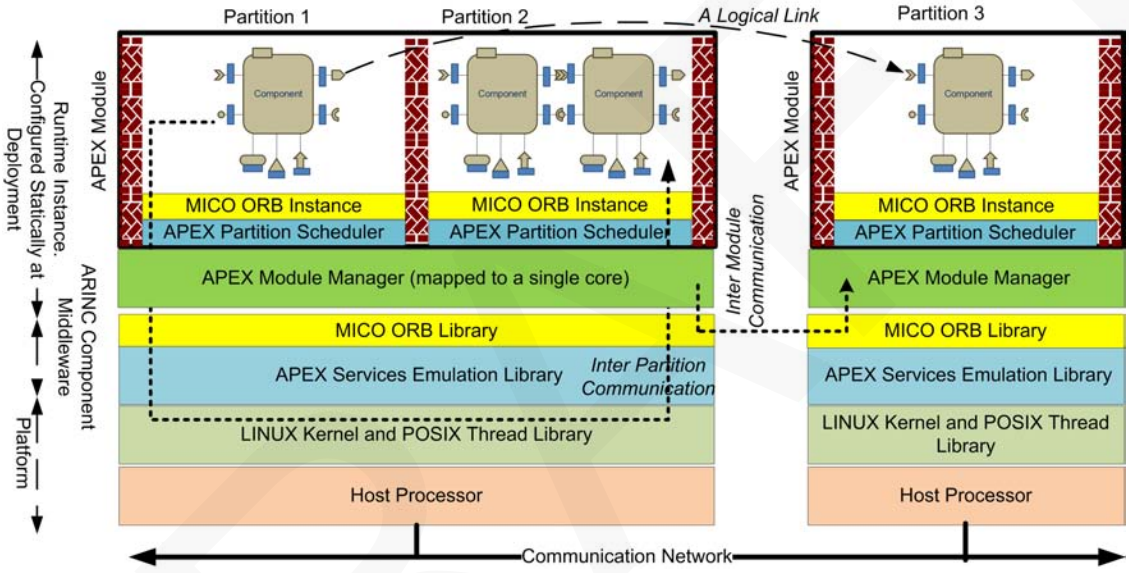

Figure 6: ARINC component framework.

Figures 6 and 8 describe the full framework depicting the ARINC component middleware, the runtime instance deployed on the middleware and a set of model-based tools for configuration, design, analysis and code-generation. Next subsections describe these layers in detail.

4.1 ARINC component middleware: Merging CCM with ARINC653

Layers of this middleware from bottom to top are described in the following subsections.

4.1.1 APEX Services Emulation Library

Due to the lack of access to a native ARINC-653 implementation, we had to first built an emulation of ARINC-653 environment. Current implementation on the Linux operating system is used for providing support for *developing* and *experimenting* with component-based systems using ARINC-653 Part 1 interfaces as described in [1]. We 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. Section A in appendix discusses certain issues faced during this exercise.

See Table 9 in the Appendix for the full list of services. These services include intra-partition process communication services, Blackboards and Buffers etc. Buffers provide a queue for passing messages and Blackboards enable processes to read, write, and clear a 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. Overall, this layer was implemented in approximately 15,000 lines of C++ code.

In this library, partitions are mapped to Linux process and ARINC-653 processes are implemented as POSIX threads. ARINC-653 processes, just like POSIX threads, share the address space. *Memory partitioning* between emulated partitions is provided by the Linux Kernel. *Temporal partitioning* is provided by a controller called module manager, discussed later in section 4.2.1.

Processes, both periodic and aperiodic, can only be created at initialization, following the ARINC-653 specification. Specified process properties include the expected deadline, 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.

4.1.2 MICO ORB Library

The next layer is MICO [34], an open source implementation of CORBA Component Model. MICO's services are used to create and host components, and perform synchronous remote method invocation (RMI). MICO's implementation of asynchronous (publish/subscribe) communication was found to be questionable as the consumer's method was invoked from within the publisher's thread using RMI calls. Asynchronous publish- subscribe communication was implemented using the inter-partition and intra-partition communication facilities in the APEX library. Next, we describe the layers that are configured and instantiated for each module in a component assembly.

4.2 Layers Instantiated for Running a Software Assembly on ARINC Component Middleware

The Component Assembly and Deployment Configuration, described in section 4.3, dictate the layers that are instantiated and configured for runtime. These layers are described next.

4.2.1 APEX Module Manager

The module manager is responsible for providing *temporal partitioning* among partitions (i.e., Linux processes). For this purpose, each module is bound to a single core of the host processor. 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 [9, 19]. Moreover, the Process periods should be multiples of respective Partition periods to reduce process jitter.

Table 3: Algorithm for Module Manager

Given: P {Set of all partitions.}
Given: $(\forall p \in P)\ PERIOD(p)$ {period of all partitions}
Given: $(\forall p \in P).\ DURATION(p)$ {duration for which p will run during each execution.}
Given: $H = LCM(period(P)$ {Hyper period value}
Given: $(\forall p \in P).\ OFFSET(p)$ {set of Offsets within a hyper period when p should start execution }
Given: $(\forall p \in P).\ EXECUTABLE(p)$ {relative path to the executable file.}
Given: $(\forall p \in P).\ SRC\_SP(p)$ {set of source sampling ports.}.
Given: $(\forall p \in P).\ SRC\_QP(p)$ {set of source queuing ports}
Given: $CHANNELS$ {inter-partition communication links. $SRC(c)$ is the source port. $DST(c)$ is the set of all destination ports.}
Require: $\sum_{p \in P} DURATION(p)/PERIOD(p) \leq 1$
Begin Module Manager
Set scheduling policy to $SCHED\_FIFO$
Set CPU affinity to a single core.
for $channel \in CHANNELS$ **do**
  for $p \in P$ **do**
    if $SRC(channel) \in SRC\_SP(p)$ or $SRC(channel) \in SRC\_QP(p)$ **then**
      $p.SRCCHANNEL.append(channel)$
    end if
  end for
end for
Initialize $OFFSETS$ {$OFFSETS$ is a sequence of tuple <time {value starting from 0}, Partition>}
for $p \in P$ **do**
  $OFFSETS.add(OFFSET(p), p)$
end for
Sort $OFFSETS$ in ascending order based on the time value.
for $p \in P$ **do**
  childprocess =fork($EXECUTABLE(p)$)
  Wait on handshake from childprocess subject to a timeout
  if timeout **then**
    Shutdown Module
    Exit
  end if
end for
for entry in $OFFSETS$ **do**
  $T1 \leftarrow currenttime$
  $T2 \leftarrow T1 + DURATION(entry.Partition)$
  Send $SIGCONT$ to $EXECUTABLE(entry.Partition)$ {SIGCONT is a POSIX signal}
  $clock\_nanosleep(T2)$ {Use a high-resolution clock such as clock realtime in Linux.}
  Send $SIGSTOP$ to $EXECUTABLE(entry.Partition)$ {SIGSTOP is a POSIX signal}
  for $c \in entry.Partition.SRCCHANNEL()$ **do**
    $c.fire()$ {Move message from source port of the channel to all the destination ports}
  end for
end for
End Module Manager

The module manager is configured with a fixed cyclic schedule. This schedule is computed from the specified partition periods and durations. It is specified as offsets from the start of the hyper period, duration and the partition to run in that window.

Computing the Module Schedule: Let $\mathbb{P}$ be the set of all partitions in a module. Let $\phi(p) \in$

$\mathbb{R}^2 \cap [0, \infty)$ denote the period of partition $p \in \mathbb{P}$. Let $\Delta(p) \in \mathbb{R} \cap [0, \phi(p)]$ denote the duration of time that a partition needs to be executed every $\phi(p)$ time units. Then hyper period H is given as $H = LCM(\phi(\mathbb{P}))$[3], where LCM is the abbreviation for the least common multiple.

Now the count of times a partition runs in a hyper period $N(p)$ is given as $N(p) = H/\phi(p)$. Let $\mathbb{O}(p) =< O_i >_{i=1}^{i=N(p)}$ be the sequence of offsets from the start of the hyper period when partition p needs to be started. We will use the notation O_i^p to denote the offset of ith execution for partition p in the hyper period.

The goal is to compute the set of all offset sequences for all partitions, $\mathbb{O}(\mathbb{P})$. A feasible valuation for $\mathbb{O}(\mathbb{P})$ can be found by solving the following system of constraints using a constraint solver library such as `Gecode`[4]:

C1 The start for all partitions must happen before the period ends i.e. $(\forall p \in \mathbb{P})(O_1^p \leq \phi(p))$.

C2 Time between any two executions should be equal to partition period i.e. $(\forall p \in \mathbb{P})(k \in [1, N(p) - 1])(O_{k+1}^p = O_k^p + \phi(p))$.

C3 The last start must finish before the hyper period ends i.e. $(\forall p \in \mathbb{P})(O_{N(p)}^p + \Delta(p) \leq H)$

C4 A partition cannot be preempted i.e. $(\forall p \in \mathbb{P})(\forall z \in \mathbb{P})(k \in [1, N(p)])(j \in [1, N(z)])$ $(O_k^p \leq O_j^z \implies O_j^z \geq O_k^p + \Delta(p))$

Once configured, the module manager implements the schedule using the `SCHED_FIFO` policy of the Linux kernel. The manager is responsible for checking that the schedule is valid before the system can be initialized i.e. all scheduling windows within a hyper period can be executed without overlap. Table 3 describes the algorithm of a module manager in pseudo code. Note that module manager is also responsible for transferring the inter-partition messages across the configured channels.

Example 2 *Figure 7 shows the example execution time line of a module with two partitions and a hyper period of 2 seconds.*

4.2.2 APEX Partition Scheduler

A partition scheduler is instantiated for each partition using the APEX services emulation library. It implements a priority preemptive scheduling algorithm using Linux `SCHED_FIFO` scheduler. The partition scheduler initializes and schedules the (ARINC-653) processes inside the partition based on their periodicity and priority. It ensures that all processes, periodic as well as aperiodic, finish their execution within the specified deadline. Upon a *hard 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. *Soft deadline violations* result in a warning issued by the middleware. The default action is to log the warning.

[2] $\mathbb{R}$ is the set of all reals

[3] Here $\phi(\mathbb{P})$ is a used as a succinct representation of set $\{x | x = \phi(p) \wedge p \in \mathbb{P}\}$. We will use this short representation for other sets also.

[4] http://www.gecode.org/

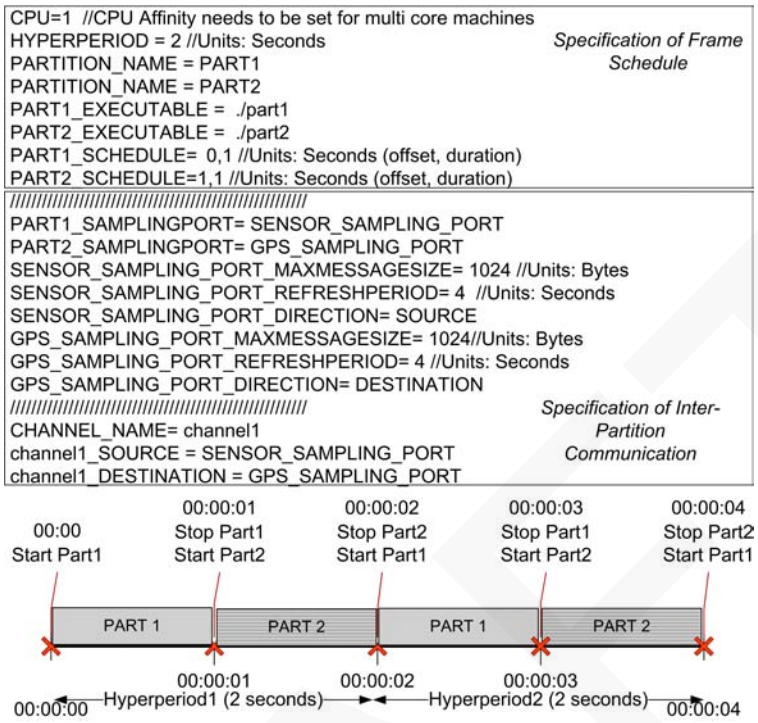

Figure 7: A module configuration and the time line of events as they occur.

4.2.3 Object Request Broker (ORB)

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 does. 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.

4.2.4 Component and Process Layers

This layer provides the glue code, generated from the definitions of components and their interfaces specified in the design environment, described in the next section. The developer is responsible for specifying the necessary process properties such as periodicity, priority, stack size, and deadline in the models. A stage of generation process uses the IDL compiler from the MICO ORB layer. Purpose of this generated code is to map the concepts of component model into the concepts exposed by the ARINC Emulator layer and the MICO ORB layer. Details about this mapping are provided in section 4.3.1. The system developer provides the functional code.

This layer also consists of component level Software Health Managers. As mentioned earlier in section 3.1, these are special processes that can take mitigation actions, if required. Readers are referred to [18, 16] for details about health managers. Next, we describe the modeling environment used to design software components and construct assemblies.

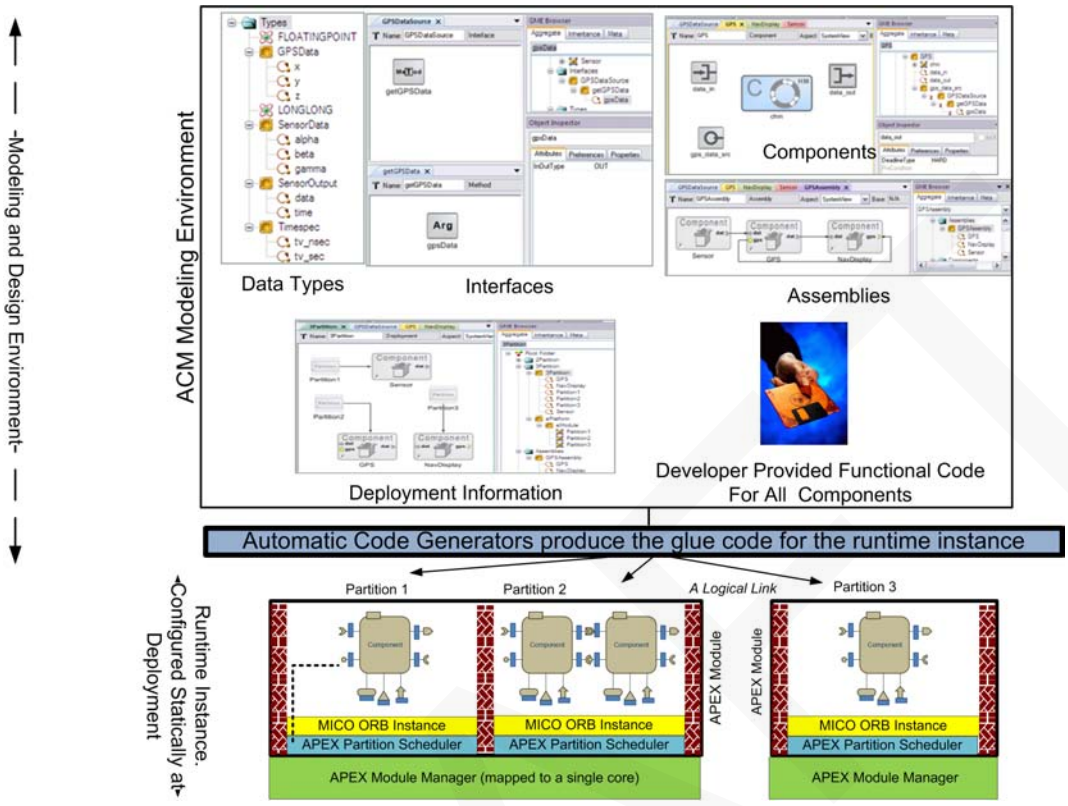

Figure 8: The system architecture, component configuration, correctness contracts, assemblies are specified in a domain specific modeling language. This model is used to generate an instance of executable assembly.

4.3 The Modeling Environment used for Design Specification

The developed framework also includes a design and modeling environment with code generation tools that can be used by component developers to model a component and the set of services that it provides independent of actual deployment configuration, see Figure 8. The grammar of this language is provided by the metamodel described earlier in Section 3.1. This environment has been created using Model Integrated Computing tools [4].

During design, each port of a component and the associated process is configured as either periodic or aperiodic. Furthermore, one must specify the deadline for the functional logic associated with the port. System integrators create software assemblies from the components, which specifies the components that are part of an application. The assembly also captures the interactions between the components through their asynchronous and synchronous interfaces. This is done by specifying a) the publisher for each consumer port, and b) the provided interface for each required interface. The Deployment model captures the mapping between the component and a partition in which it resides. The modeling environment enforces certain design rules to ensure the compatibility between ports of interacting components and correctness of design. See Table 4 for a summary of these rules. Finally, the code generator is used to produce non-functional glue code to deploy the software assembly on the platform.

Example 3 *Figure 9 shows the modeled assembly for the example described in section 1 . The*

Table 4: List of design constraints enforced by the ACM modeling and design environment

1	A Required interface in an assembly should be connected to a provided interface
2	Components in assembly cannot have publishers and consumers with unspecified types
3	For HARD Deadline tasks, Deadline should be less than or equal to period
4	Deadline value should be > 0 and $< \infty$.
5	A consumer in an assembly should have an event source i.e. it should be associated with a Publisher.
6	Name cannot clash with the reserved keywords. This match is case-insensitive.
7	A method defined inside an interface cannot be periodic
8	All names must be a valid identifier i.e. they must match the regular expression `"^[a-zA-Z][_a-zA-Z0-9]*$"`
9	One way call cannot have out parameters
10	A component member cannot have the same name as its container.
11	Component Trigger should be periodic
12	Partition duration should be either > 0 and $< \infty$. -1 indicates aperiodic
13	Partition period should be either > 0 and $< \infty$. -1 indicates aperiodic
14	Sum of Duration/Period for all Partitions belonging to module should be less than 1
15	Names of a component members must be unique
16	The interface type of a required interface port must match the interacting provided interface.
17	Deadline of a require interface port must be greater than or equal to the deadline of the provided interface port
18	Consumer validity cannot be less than or equal to 0

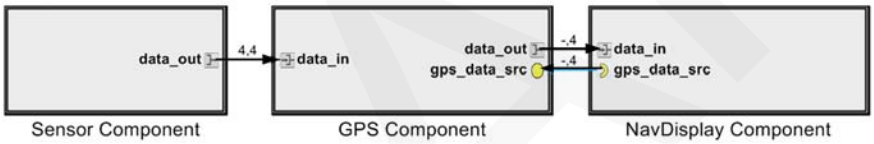

Figure 9: Example: Component Interactions. Here each interface is annotated with its (periodicity, deadline) in seconds.

system integrator can choose to deploy a developed assembly across different partitions and across different modules. Some of these decisions are based on the level of isolation required between the components. Based on the given deployment, the necessary configuration information is generated. Figure 12 shows deployment on a module with two partitions.

Next section describes how different concepts in modeling language are mapped to the underlying ARINC Component middleware layers.

4.3.1 Generating Runtime Layers from Models

This section describes the association between the model entities and the run-time. Each module in the design specification is mapped to an instance of the Module Manager, which launches and schedules the Partitions on the module. The module manager creates each ARINC partition as a Linux process. Inside each partition, an aperiodic ARINC process is created to host the ORB. The partition hosts the component home.

The code generator synthesizes the code and the necessary IDL files so that one can use the services provided by the MICO CCM implementation to create a singleton instance of each component, inside the partition. Singleton instance implies that we disable the session life cycle of components – they cannot have more than one instance in the same partition. Each component

Table 5: Implementation of concepts in the ACM middleware. (RMI=Remote Method Invocation).

Concept	**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 Trigger	Periodic	Periodic process, mapped to an OS Thread	Process start, stop
Asynchronous ports	Periodic	Periodic process, mapped to an OS Thread	Process start, stop
Ports	APeriodic	APeriodic process, mapped to an OS Thread.	Process start, stop
Component Locks	N/A	APeriodic process, mapped to an OS Thread.	Semaphores
Synchronous RMI	Both Aperiodic, and collocated	Caller method signals callee to release then waits for callee until completion.	Event, Blackboard
Synchronous RMI	Both Aperiodic, and non-collocated	Caller method sends RMI to release callee then waits for RMI to complete.	TCP/IP, Semaphore, Event
Asynchronous Publish-Subscribe	Periodic consumer and Collocated	Callee is periodically triggered and polls event buffer (Blackboard) - validity flag indicates whether data is stale or fresh	Blackboard
Asynchronous Publish-Subscribe	Periodic consumer and 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 consumer and Collocated	Callee is released when event is available	Blackboard, Semaphore, Event
Asynchronous Publish-Subscribe	Aperiodic consumer and Non-collocated	Module manager moves the message. Callee is released upon receipt by an aperiodic trigger process	Queuing Port, Channel

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 generated code executes an ORB request for a particular interface by releasing the appropriate ARINC process (i.e. OS thread). Multiple processes belonging to the same component may engage read/write locks depending on whether they are marked as read-only or not. These locks are implemented using ARINC-653 semaphores. This is required to ensure that race conditions do not occur. Note that all the lock manipulations are in the generated code.

Remark 5 *Our current implementation assigns priorities to generated runtime schedulable entities automatically, as follows: Priority of Module manager > Priority of Partition Scheduler > Priority of ORB > Priority of any health management ARINC process > Priority of all other ARINC processes. All component processes are assigned same priority. However, this can be changed manually in the generated code.*

Table 5 summarizes how different concepts from component model are implemented in the ACM middleware. A Periodic publisher is bound to an ARINC process. The real time properties

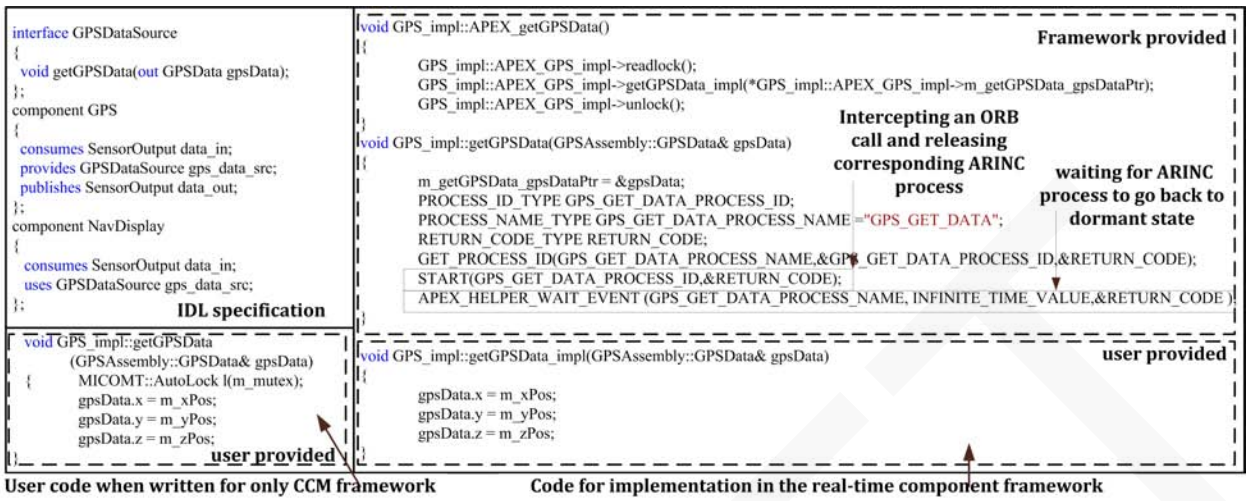

Figure 10: Equivalent implementation of a CORBA CCM interface in our ACM framework.

of this process i.e. the deadline and periodicity are determined from the values specified in the model.

See Appendix B for details on code generation process for periodic publishers, aperiodic publishers, periodic consumers, aperiodic consumers, and synchronous ports.

Example 4 *Figure 10 shows a portion of the IDL generated for the assembly shown in Figure 9. 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 generated 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 until the corresponding aperiodic process finishes by using the wait call on an APEX event used for notification purposes.*

The interactions between the synchronous ports (requires and provides) is implemented using the standard remote method invocation (RMI) capability available in the CCM / CORBA. It is initiated by invoking a function (on the requires / receptacle side) that launches the ARINC process associated with appropriate interface-method. An RMI call is issued from the ARINC process. On the side of the provided interface port, the RMI call is processed by starting and waiting for the completion of the ARINC Process associated with the interface-method. Once the ARINC process finishes, the RMI chain completes by transmitting the result via the standard CORBA/CCM mechanisms.

The interactions between the asynchronous ports (publisher and consumer) are initiated when the ARINC process associated with the publisher either is started by the scheduler (periodic) or is started due to an explicit invocation (aperiodic). The publisher process assembles and publishes the event, which is communicated to each of consumers. *The generated transport mechanism for a publish interaction depends on the location of the Consumer Process relative to the publisher (collocated or not) and the nature of the Consumer (Periodic/Aperiodic).*

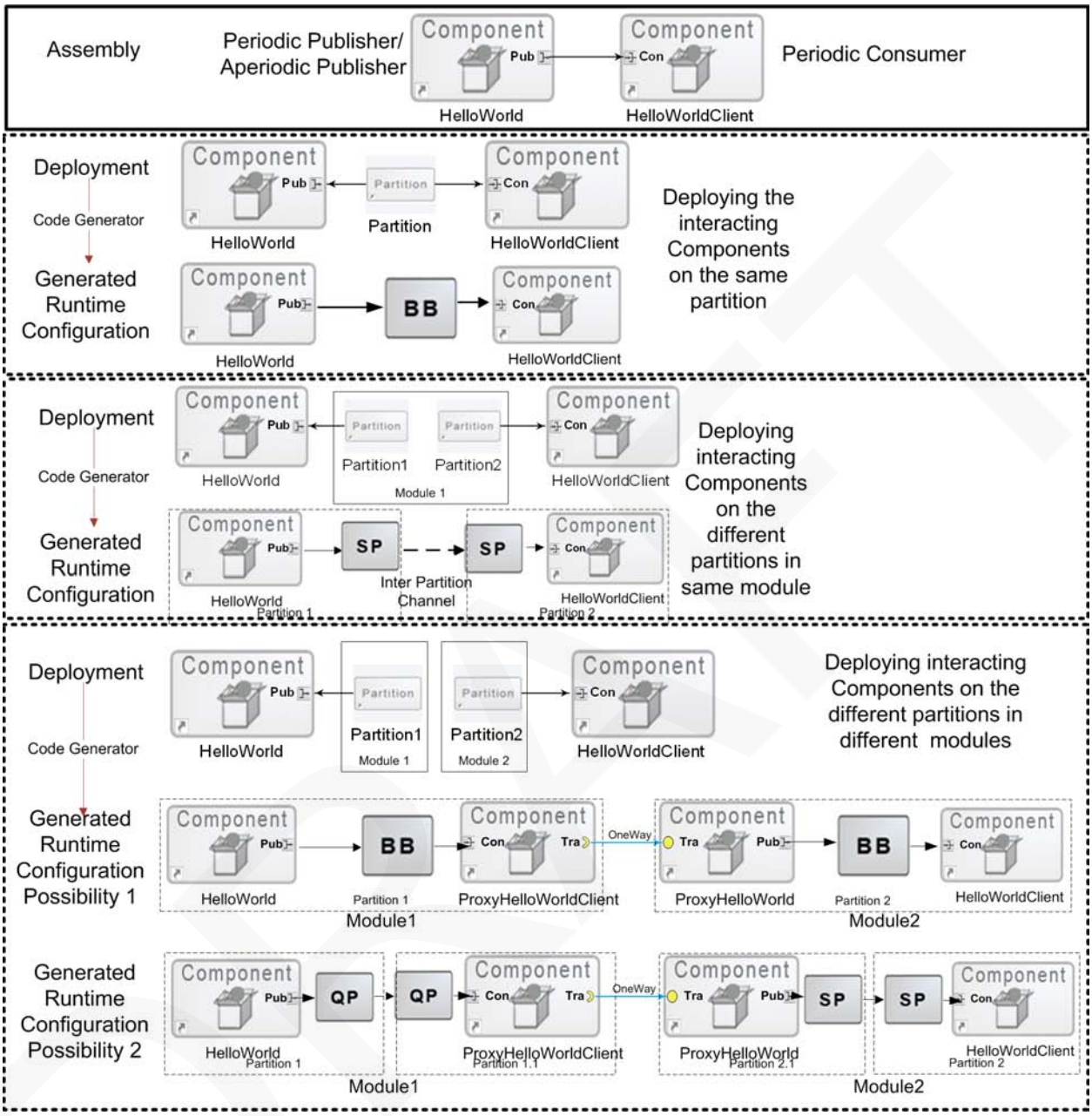

Figure 11: Same interaction results in different runtime configurations based on deployment. This is done by the code generator, freeing the system developer to focus on the application logic instead.

Intra-Module Interaction For example, Figure 11 summarizes different generation scenarios for interaction between a periodic publisher and periodic consumer. If the Publisher and Consumer processes were in different Partitions (OS Processes) i.e. were non-collocated, then the event transmission maps to an ARINC communication via sampling ports (for Periodic Consumer) or queuing ports (for Aperiodic Consumer). When the Publisher and Consumer ARINC processes are in the same Partition (OS Process, i.e. co-located), the event transmission maps to an ARINC

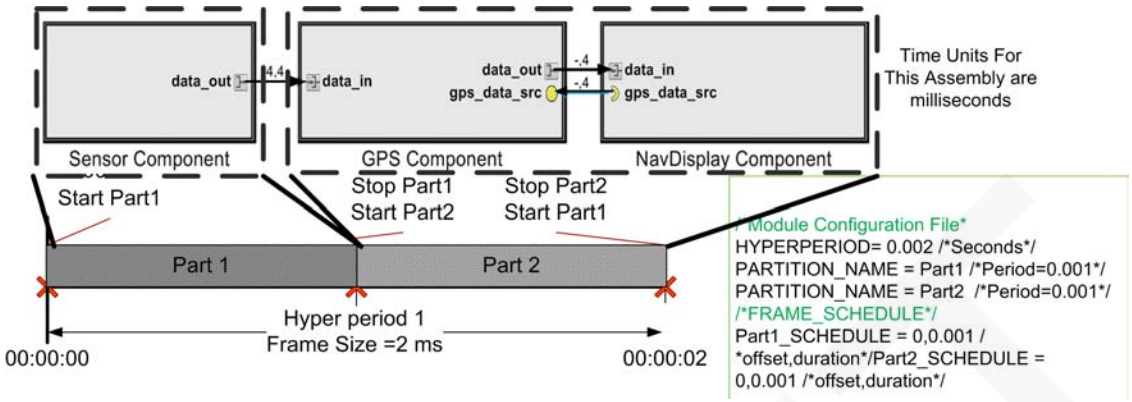

Figure 12: Software Assembly and Deployment Diagram used in the case study.

communication via blackboard (for Periodic Consumer) and buffer (for Aperiodic Consumer). If the Consumer is periodic, the associated ARINC Process is started at a specific rate by the scheduler. The ARINC process (for Periodic Consumer) reads the associated Sampling port or Blackboard and operates on the event. In the interaction between asynchronous ports on different partitions (non-co-located), the Module Manager is responsible for transferring the event generated and stored in the sampling or queuing port of the publisher to the appropriate sampling or queuing port of the consumer. It does this whenever any partition finishes its cycle (see Table 3).

Inter-Module Interaction Inter module communication is an area which has not been covered clearly in the ARINC-653 specification [1]. One way to implement the interactions between publisher and consumer across modules is to use additional proxy components with a one-way synchronous interaction between them. The proxy components, shown in bottom section of Figure 11 are automatically generated. However, the timing of message communication depends upon the deployment of proxy components. For example, if the ProxyHelloWorldClient is collocated with HelloWorld and ProxyHelloWorld is collocated with HelloWorldClient, then the message transmission from publisher to consumer port is done immediately, as soon as the message is published. However, if the ProxyHelloWorldClient is put in a separate partition on the same module with HelloWorld and ProxyHelloWorld is is put in a separate partition on the same module with HelloWorldClient, the real transmission of message from ProxyHelloWorldClient to ProxyHelloWorld will be governed by the schedule set for partition1.1. The actual receipt of message at the consumer port of HelloWorldClient will be governed by the schedule of partition2.2. Thus, it is possible to obtain a time triggered messaging between modules by setting a synchronized schedule for partitions of ProxyHelloWorldClient and ProxyHelloWorld.

5 Case Study

Figure 12 shows the assembly discussed earlier in section 1 deployed on a module with two partitions. Connections between two ports have been annotated with the (periodicity, deadline) in milliseconds on the downstream port. Table 6 summarizes all the ports.

Figure 12 also describes the periodic schedule followed by the partitions, overseen by Module manager. In this example, Partition 1's phase was 0 milliseconds, while its duration was 1 millisecond. Partition 2's phase was set to 1 millisecond. Its duration was also 1 millisecond. This

Table 6: Component ports defined in the assembly diagram.

Period(secs)	Component	Port	WCET(secs)
0.004	Sensor	data_out	0.004
0.004	GPS	data_in - after processing sends an event to Display	0.004
Sporadic	GPS	gps_data_src.GetGPSData	0.004
Sporadic	NavDisplay	data_in - after processing calls GetGPSData	0.004
Sporadic	NavDisplay	gps_data_src.GetGPSData	0.004

Table 7: Processes created by the framework for the ports defined in Table 6 and Figure 12.

Partition	Process Name	Associated port	Period	Deadline	Type
Part 1	Part1 ORB Process		Aperiodic	Infinite	SOFT
Part 1	Sensor.DataOut	data_out	4msec	4msec	HARD
Part 2	Part2 ORB Process		Aperiodic	Infinite	SOFT
Part 2	GPS.DataIn	data_in	4msec	4msec	HARD
Part 2	NavDisplay.DataIn_T (trigger)	data_in	Aperiodic	Infinite	HARD
Part 2	NavDisplay.DataIn	data_in	Aperiodic	4msec	HARD
Part 2	NavDisplay.U_NavDisplay_getGPSData	gps_data_src	Aperiodic	4msec	HARD
Part 2	GPS.P_GPS_getGPSData	gps_data_src	Aperiodic	4msec	HARD

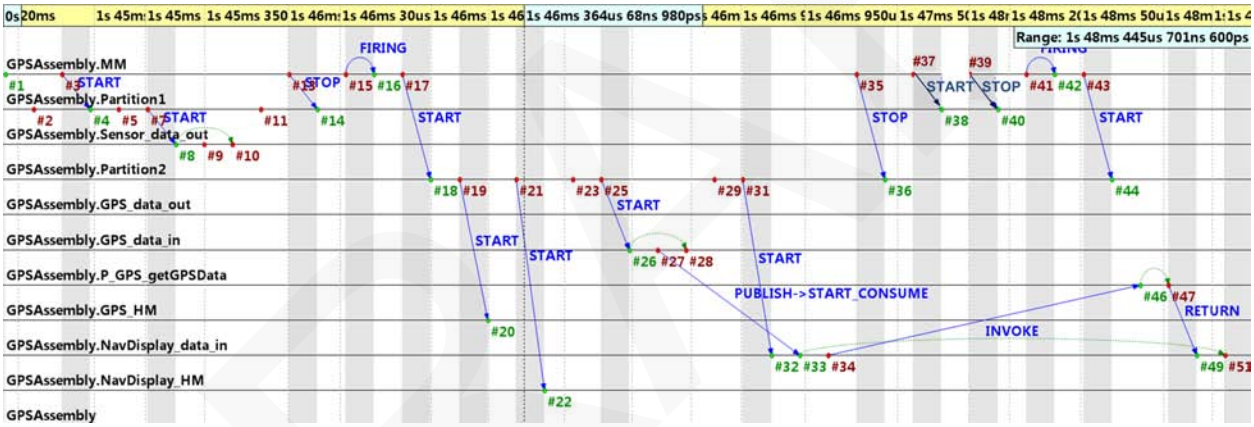

Figure 13: Sequence of Events for the case study. The scale is non-linear.

ensured that both partition got 1 millisecond of execution time every 2 milliseconds. The logical publish-consume connection between sensor and GPS components is implemented via an ARINC-653 sampling port. A Channel connects the source sampling port from partition 1 to destination sampling port in partition 2.

Table 7 shows all the ARINC-653 processes (and the partition they were deployed on) required to build this example. To implement this particular example, the developer had to write 148 lines of code, while 1027 lines of code were generated.

Figure 13 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++ [5]. 0^{th} event marks the start of the module manager, which then creates the Linux processes for the two partitions. Each partition then creates its respective (APEX) processes and signals the module manager. This all happens before the frames are scheduled. After the occurrence of 0^{th} event, module manager signals partition 1 to start. Upon start, partition 1 starts the ORB process that handles all CORBA-related functions. It then starts the sensor health manager. Note that all processes are started in

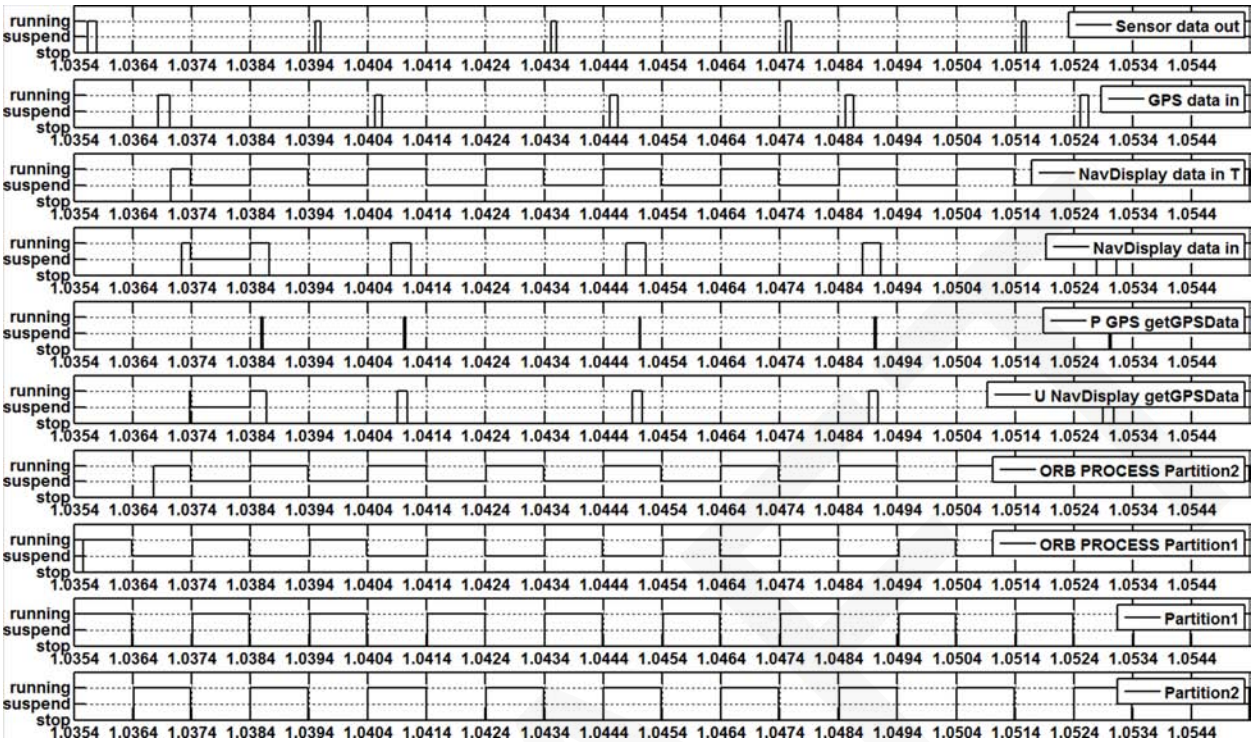

Figure 14: Timing chart for all processes and partitions. Time on x-axis is in seconds.

an order based on priority. It starts the periodic sensor process at event number 8. The sensor process publishes an event at event number 9 and finishes its execution at event number 10. After 1 millisecond since its start, partition 1 is stopped by the module manager at event number 14. Immediately afterwards, partition 2 is started. Partition 2 starts all its ORB process and health managers at the beginning of its period. At event 26, partition 2 starts the periodic GPS consumer process. It consumes the sensor event at event 27. At event 27, GPS publisher process produces an event and finishes its execution cycle at 28. The production of GPS event causes the sporadic release of aperiodic consumer process in Navigation Display (event 33). The navigation process uses remote procedure call to invoke the GPS get data ARINC process. The GPS data value is returned to navigation process at event 49. It finishes the execution at event 51. Partition 2 is stopped after 1 millisecond from its start. This marks the end of one frame. Note that these events do not capture the internal functional logic of the GPS algorithm.

Figure 14 shows the timing diagram for all the processes executed in this example. Note that partition 1 and partition 2 are temporally isolated. Also, note that when a partition, e.g. partition 1, is put out of context, all running processes in that partition are suspended. Table 8 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. This value was calculated as the shift from pre-calculated schedule. Notice that the standard deviation is low. This implies that the jitter in the system is low.

That means after the initial shift due to time taken during initialization, the gap between any two consecutive executions almost remained same.

Table 8: Summary of observed jitter. Calculated as shift from pre-calculated schedule.

Process	Std (μs)	Mean (μs)	Max (μs)
Part 1	2.06	8.24	11.31
Part 2	4.10	13.97	22.98
GPS.DataIn	0.84	329.38	329.93
Sensor.DataOut	2.94	130.78	133.95

6 Discussion and Summary

6.1 Differences between ARINC Component Model (ACM) and CORBA Component Model (CCM) Approaches

The differences between the two approaches exist on three different levels: model, mapping, and implementation. While CCM supports attributes, ACM does not; ACM allows definition of state variables (internal to the component by default) which can be observable from outside the component. The Ports (or external interfaces) in ACM are extended with additional attributes - period, deadline, deadline-type, pre-condition, post-condition etc. Further, ACM allows additional functional entities such as Component Triggers (periodic methods internal to the component), monitor interfaces (to monitor the component's resource usage), health managers (to detect and react to anomalous behavior).

Some differences also exist in implementation and mapping of the modeling entities to runtime. In CCM implementations, the functional logic belonging to an interface port is executed on a new, dynamically created, or pre-existing but dynamically released worker-thread. In ACM, each port has a dedicated ARINC process (configured during initialization) where the functional logic is executed. 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. Further, the port attributes (period, deadline etc.) are used to configure the ARINC process as periodic or aperiodic.

The non-availability of dynamic allocation and the requirement for static binding of the component ports to ARINC processes in ACM ensures that there is only a single instance of every Component in the Assembly, created during initialization. Further, the generated code ensures thread synchronization between the external ports of the components as they are launched on separate ARINC Process (mapped to OS Thread). Standard CCM implementations permit session-based configurations where new instances of the Component are created per session. The thread-synchronization between the multiple interfaces of a component is handled at the object level in CCM, the dynamically released worker threads need to get access to the component object before execution. In our implementation, synchronization between component ports is handled by the generated code using read/write locks implemented using ARINC-653 semaphores.

6.2 Lessons Learned

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.

Furthermore, 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, framework provides the initialization code 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 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.

Finally, we restrict only one process to access a component's state at a time to maintain consistency 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.

6.3 Extending CCM

Our component model presented in Section 3 is an extension of the standard CORBA Component Model. 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.

The CORBA interceptors could be used to implement the runtime monitors that check the pre-conditions and post-conditions for each port. 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 is 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 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.

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 an interface provided by a component is read-only.

6.4 Problems Identified

During our experiments, we came across issues that are important to emphasize. Foremost, we discovered that in typical CCM implementations like MICO [34] and CIAO [12], the method used by

an event source to publish an event is implemented as a two-way blocking call, which is not asynchronous. In other words, the publisher's thread will invoke the subscriber's or the event broker's consume methods in the same thread. Due to this, we had to implement the event-based communication mechanism through Blackboards and Buffers provided by our APEX library, where the publisher and the subscriber use separate threads. Inter-partition event-based communication was mapped to sampling and queuing ports. Inter-module communication was mapped to queuing ports and one-way calls.

We identified some problems with the ARINC 653 specification as well. For example, ARINC-653 stipulates that aperiodic processes can extend their deadline using *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.

Finally, ARINC-653 specification does not permit changing the properties of an ARINC process (e.g. relative deadline[5]) at runtime. This decision is primarily governed by the analysis advantages that such static configuration provides. Due to this static configuration constraint, and given that different ports of an ACM component have different real-time properties, it is not possible to use a pool of processes and dynamically assign them to the ports. Therefore, 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 in a large number of processes for a bigger software assembly.

7 Related Research

7.1 Real Time Software Component Frameworks

An approach to objects based on time-triggered (periodic) and event-triggered (aperiodic) methods has been presented in [26]. 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 [40], ControlShell [36], eCos [20] , Koala [41] 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) [33] 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

[5] relative to the start time of the process

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 [7].

CIAO [12] and PRISM [38] 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 [27]. 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 enforce similar restrictions in our middleware to keep the component interactions simple and predictable. While CAmkES 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) [14]. POK is the runtime for ARINC-653 Annex of the Architecture Analysis and Design Language (AADL) [22]. AADL provides constructs for modeling software and hardware components. Note that these software components are not the same as a reusable software component as described in the CORBA component Model. Rather here 'components' is a term used to refer to various units of software functions, software data, threads and processes. Hardware components include processors, buses, memory and devices. AADL models also specify interaction between these components. Properties such as deadline, worst case execution time, critical for assessing the performance and functionality of the system can also be specified in AADL. POK uses the OCARINA[6] framework to automatically configure and deploy processes and partitions. The main difference between this approach and our work is the level of abstraction at which system is designed. While in our approach, system is designed using high-level components and high-level synchronous and asynchronous interactions, in POK systems are designed at the level of individual ARINC-653 processes and all interactions are specified at the level of native ARINC-653 mechanism.

DIANA [37] 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 [28]. 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

[6] http://ocarina.enst.fr/

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.

7.2 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 [9]. They showed that partitions could 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. [19]. 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 [30]. 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 [11] 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 [8]. 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 therefore it is possible that deadlines are violated during run-time. Therefore, the ACM middleware actively monitors all deadlines. Any violation results in the default action of stopping the faulty process. An API is available to restart the faulty Process after a reset.

7.3 Design by Contract

Formal argument for checking correctness of execution of a computer program based on a first order logic system was first presented by Hoare in [24]. Later this concept was extended to distributed systems by Meyer in [31, 25]. A contract implemented by Meyer specified the requires and ensure clauses as assertions specified by a list of boolean expressions. These assertions were specified as logic operations upon the value domain of the program variables and were compiled out in the running system.

Conmy et al. presented a framework for certifying Integrated Modular Avionics applications build on ARINC-653 platforms in [13]. 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 specification of these constraints.

In ACM, these correctness conditions are specified by pre-conditions and post-conditions, which can be defined over both the value-domain and temporal domain of program variables as well as the state variables belonging to the component. We envision that these checks are performed in real time on the system. This is especially necessary because there is a high likelihood for software defects being present in complex systems that arise only under exceptional circumstances. These circumstances may include faults in the hardware system (including both the computing and non-computing hardware); software is very often not prepared for hardware faults [18].

8 Conclusion

In this paper, we described the design and implementation of a framework that enables system developers to construct real time software as a composition of well-defined components. The facilities discussed in the ARINC component model define specific timing, periodicity and correctness criteria with each component port. The framework provides well-defined interactions, both synchronous and asynchronous. Mixing these interactions with periodic and aperiodic triggering of ports provides the system developers a broad range of well-defined compositional semantics. Finally, the deployment model allows the developers to partition the software system both spatially and temporally.

To implement this framework we combined CCM with ARINC-653, which provides partitioning capability. We created an ARINC-653 emulator 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 compatibility issues between CCM and ARINC-653. These differences lead us to recognize that further developments are needed that integrate components with a hard real time platform.

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 [7].

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 LaRC for their help and guidance on the project.

References

[1] Arinc specification 653-2: Avionics application software standard interface part 1 - required services. Technical report.

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

[3] Model-Integrated Computing, http://www.isis.vanderbilt.edu/research/mic.

[4] The ISIS Model Integrated Computing (MIC) Toolsuite.
[5] OMNeT++ Network Simulator.
[6] Joint advanced strike technology program, avionics architecture definition appendix B. Technical report, August 1994.
[7] 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.
[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
[9] N. Audsley and A. Wellings. Analysing APEX applications. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 39, 1996.
[10] K. Balasubramanian, N. Wang, and D. C. Schmidt. Towards composable distributed real-time and embedded software. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:226, 2003.
[11] 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.
[12] CIAO. http://download.dre.vanderbilt.edu/.
[13] P. Conmy, J. McDermid, and M. Nicholson. Safety analysis and certification of open distributed systems. In *International System Safety Conference,*, Denver, 2002.
[14] 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, Berlin, Heidelberg, June 2009. Springer-Verlag.
[15] N. Diniz and J. Rufino. ARINC 653 in space. In *Data Systems in Aerospace*. European Space Agency, May 2005.
[16] 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`.
[17] A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. A real-time component framework: Experience with ccm and arinc-653. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, pages 143–150, 2010.
[18] A. Dubey, G. Karsai, and N. Mahadevan. Towards model-based software health management for real-time systems. Technical Report ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University, August 2010.
[19] 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/`.
[20] eCos. http://ecos.sourceware.org/.
[21] V. Fay-Wolfe, L. C. DiPippo, G. Copper, R. Johnston, P. Kortmann, and B. Thuraisingham. Real-time corba. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1073–1089, 2000.
[22] P. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the sae architecture analysis design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering. In *Architecture Description Languages*, volume 176 of *IFIP International Federation for Information Processing*, pages 3–15. 2005.
[23] A. Goldberg and G. Horvath. Software fault protection with ARINC 653. In *Proc. IEEE Aerospace Conference*, pages 1–11, March 2007.
[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
[25] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[26] K. Kim. Object structures for real-time systems and simulators. *Computer*, 30(8):62–70, Aug 1997.
[27] 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.
[28] 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.
[29] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.
[30] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2):257–269, 2005.
[31] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
[32] D. Morgan. Pave pace: system avionics for the 21st century. *Aerospace and Electronic Systems Magazine, IEEE*, 4(1):12 –22, Jan. 1989.
[33] R. Obermaisser and H. Kopetz, editors. *Genesys An Artemis Cross-Domain Reference Architecture For Embedded Systems*. Sudwestdeutscher Verlag fur Hochschulschriften AG, 2009.
[34] A. Puder. MICO: An open source CORBA implementation. *IEEE Softw.*, 21(4):17–19, 2004.
[35] J. Rushby. Modular certification. Technical report, Sept. 2001.
[36] 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.
[37] 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.
[38] M. Schulte. Model-based integration of reusable component-based avionics systems - a case study. In *ISORC 2005*, pages 62–71.
[39] V. Subramonian, G. Deng, C. D. Gill, J. Balasubramanian, L.-J. Shen, W. Otte, D. C. Schmidt, A. S. Gokhale, and N. Wang. The design and performance of component middleware for qos-enabled deployment and configuration of dre systems. *Journal of Systems and Software*, 80(5):668–677, 2007.
[40] Tinyos. http://webs.cs.berkeley.edu/tos/.
[41] 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.
[42] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electron. Notes Theor. Comput. Sci.*, 141(3):53–71, 2005.
[43] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the corba component model. *Component-based software engineering: putting the pieces together*, pages 557–571, 2001.
[44] C. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, pages 2.A.1–1 –2.A.1–10, oct. 2007.

A ARINC-653 Emulator on Linux

Current implementation has been tested on 2.6.9 and above kernels. Table 9 lists the implemented services. However, we have noticed that the scheduling jitter is less if the emulator is used on a kernel older than 2.6.21. This is because of the introduction of high resolution timers and tick-less kernel since that version. Some issues that we discovered during this work are:

- Linux real time priorities range from 1 to 99. However, ARINC-653 priorities range from 1 to 63.
- Running FIFO Scheduler needs real-time privileges. The simplest way to achieve this is to use the root account.

- Default message queue size in Linux is typically limited to 10. Creating a queue with size greater than 10 will cause problems. This is easily remedied by changing the sys config parameters.
- One of the ARINC-653 API is the STOP call, which can stop an ARINC process from execution and return it to the entry point of the function being executed in the process. The only way to achieve this in Linux with Pthreads is to terminate the thread and relaunch a new thread with the same entry point as before. This specifically becomes a problem if the executing function does not include interrupt points such as pthread cancel.

Table 9: List of ARINC-653 services implemented in the ARINC Emulation Layer. Refer to [1] for further detail on these interfaces.

Category of Services	**Provided**	**Service Description**
Event Management Services	CREATE_EVENT	Create an event
	SET_EVENT	Set and event to notify a waiting process
	RESET_EVENT	Clear the state of the event
	WAIT_EVENT	Block on an event and wait for notification
	GET_EVENT_ID	Retrieve the identifier for an event using a unique name
	GET_EVENT_STATUS	Determine the number of processes waiting on an event
Semaphore management services	CREATE_SEMAPHORE	Create a semaphore for synchronization
	WAIT_SEMAPHORE	Block on the semaphore
	SIGNAL_SEMAPHORE	Post to the semaphore
	GET_SEMAPHORE_ID	Determine the platform identifier for it
	GET_SEMAPHORE_STATUS	Get the number of resources available and number of waiting processes
Partition management services	GET_PARTITION_STATUS	Determine if the partition is in the Normal State or Idle State
	SET_PARTITION_MODE	Set the Partition to Normal upon initialization
Process management services	CREATE_PROCESS	Create an ARINC process and set its real-time properties.
	SET_PRIORITY	Set the priority of a specific process.
	SUSPEND_SELF	Suspend self and yield resources.
	SUSPEND	Suspend a specific process
	RESUME	Resume a previously suspended process
	STOP_SELF	Set self-state to dormant. After this the process will not run until restarted.
	START	Start a given process.
	DELAYED_START	Start a process after the specified delay.
	GET_PROCESS_ID	Determine the unique identifier for a process.
	GET_MY_ID	Determine your own identifier.
	GET_PROCESS_STATUS	Determine the state of a process, whether it is running or not.
Time management services	TIMED_WAIT	Wait or suspend execution for the specified time.
	PERIODIC_WAIT	Wait till the next periodic cycle of the process.
	GET_TIME	Find current system time in nanoseconds
	REPLENISH	Extend the deadline for the current process
Blackboard management services	CREATE_BLACKBOARD	Create an intra-partition blackboard
	DISPLAY_BLACKBOARD	Write to the blackboard. This overwrites past value.
	READ_BLACKBOARD	Read the entry from the blackboard. If the blackboard is empty, reader can choose to be blocked for the specified time.

continued on next page

continued from previous page		
Category of Services	**Provided**	**Service Description**
	CLEAR_BLACKBOARD	Erase the contents of the blackboard.
	GET_BLACKBOARD_ID	Determine the identifier for a blackboard.
	GET_BLACKBOARD_STATUS	Get status, number of bytes written and number of processes waiting.
Buffer management services	CREATE_BUFFER	Create a buffer.
	SEND_BUFFER	Write to a buffer. Process blocks for the specified time if the buffer is full.
	RECEIVE_BUFFER	Read from the buffer. Process blocks for the specified time if the buffer is empty.
	GET_BUFFER_ID	Get Buffer identifier.
	GET_BUFFER_STATUS	Get Buffer Status
Sampling Port management services	CREATE_SAMPLING_PORT	Create a Sampling Port.
	WRITE_SAMPLING_MESSAGE	Write to the sampling port. This overwrites past data.
	READ_SAMPLING_MESSAGE	Read the data from the port. The call also returns a flag stating if the data is stale.
	GET_SAMPLING_PORT_ID	Get the port identifier.
	GET_SAMPLING_PORT_STATUS	Get status of sampling port.
Queuing Port management services	CREATE_QUEUING_PORT	Create a queuing port.
	SEND_QUEUING_MESSAGE	Write to the port. This call blocks for the specified time if the queue is full.
	RECEIVE_QUEUING_MESSAGE	Read from the port. This call blocks for the specified time if the queue is empty.
	GET_QUEUING_PORT_ID	Get the port identifier.
	GET_QUEUING_PORT_STATUS	Get the port status.

B Code Generation Templates for Component Ports

B.1 Code Template for Publishers

This subsection describes the code template that is used to generate the publisher logic. Upon the start of a cycle, the process tries to acquire a write lock to the component. This is necessary to ensure that at a time only one thread is active in the component. Each publisher is statically given a list of ports on which it publishes. The partition scheduler triggers this process periodically and monitors it for deadline violation. During each cycle, the publisher process calls the user provided functional code `USER_FILL_PUBLISHER_NAME_DATA(data)` and then publishes the data. The pre-conditions and post-conditions are validated based on provided specifications to ensure that the publisher is working correctly. Aperiodic publishers are also bound to one ARINC process. The generated code for aperiodic publisher is similar to the periodic publisher. The only difference is that we also generate a wrapper function call that can be called from any other process inside the component.

Given: Name {Name of the publisher}
Given: E {Set of All Event Ports. An event port is an abstract concept. It maps to either a sampling port or a queuing port depending whether the consumer is periodic or aperiodic.}
Given: COMP {implementation class for the component}
Begin PeriodicPublisher {Partition Scheduler starts the deadline violation monitor}
1: COMP→writelock($DeadlineTime$,$return\_code$)
2: **if** $return\_code$ is TIMEOUT **then**
3: **RAISE** (LOCK TIMEOUT ERROR)
4: **return**
5: **end if**
6: Check pre-conditions

7: **if** pre-conditions not met **then**
8: **RAISE** (pre-condition Violation)
9: **return**
10: **end if**
11: Initialize(data) {data to be published}
12: USER_FILL_PUBLISHER_NAME_DATA(data) {User function details are filled in by the system developer. Name is replaced with the actual value upon code generation.}
13: **if** user exception occurred **then**
14: **RAISE** (user exception)
15: **return**
16: **end if**
17: Initialize($failedpubs$)
18: **for** e ∈ E **do**
19: e.publish(data,return_code)
20: **if** return_code is not NO_ERROR **then**
21: $failedpubs$.insert(e)
22: **end if**
23: **end for**
24: **if** $failedpubs$.size() is not 0 **then**
25: **RAISE** (failedpubs)
26: **return**
27: **end if**
28: Check Post-conditions
29: **if** Post-conditions not met **then**
30: **RAISE** (Post-conditions Violation)
31: **return**
32: **end if**
33: COMP→unlock()
34: **return**
End PeriodicPublisher {Partition Scheduler stops the deadline violation monitor}

B.2 Code Template for Periodic Consumer

The generated code for periodic consumers looks similar to the code generated for periodic publisher. A periodic consumer is bound to an ARINC-653 sampling port. Upon reading a data item, the process checks its validity and raises an error if the data is stale (i.e. invalid). Then, control is transferred to the user code using `USER_HANDLE_CONSUMER_NAME_DATA(data)`. The software developer supplies the function body.

Given: Name {Name of the consumer}
Given: SP { A Sampling port.}
Given: COMP {implementation class for the component}
Begin Periodic Consumer {Partition Scheduler starts the deadline violation monitor}
1: COMP→writelock($DeadlineTime$,$return\_code$)
2: **if** $return\_code$ is TIMEOUT **then**
3: **RAISE** (LOCK TIMEOUT ERROR)
4: **return**
5: **end if**
6: Initialize(data) {data to be read}
7: SP.read(data) {SP is a sampling port}
8: validity ← $age(data) \leq validity\_period$
9: **if** validity==false **then**
10: **RAISE** (Validity Violation)
11: **return**

12: **end if**
13: Check pre-conditions
14: **if** pre-conditions not met **then**
15: **RAISE** (pre-condition Violation)
16: **return**
17: **end if**
18: USER_HANDLE_CONSUMER_NAME_DATA(data) {User function details are filled in by the system developer. Name is replaced by the actual name upon generation}
19: **if** user exception occurred **then**
20: **RAISE** (user exception)
21: **return**
22: **end if**
23: Check Post-conditions
24: **if** Postconditions not met **then**
25: **RAISE** (Post-conditions Violation)
26: **return**
27: **end if**
28: COMP→unlock()
29: **return**
End Periodic Consumer {Partition Scheduler stops the deadline violation monitor}

B.3 Code template for aperiodic consumer and its associated trigger method

For aperiodic consumers, an additional ARINC Process is created to wait for new events and trigger the actual Consumer Process (when a new event is available).

Given: QP { A Queuing port.}
Given: COMP {implementation class for the component}
Begin Aperiodic Consumer Trigger Process {This is an infinite deadline process}.
1: **loop**
2: Initialize(data) {data to be read}
3: QP.read(data) {Poll the port. This call will block if there is no data in the queuing port}
4: validity ← $age(data) \leq validity\_period$
5: **if** validity==false **then**
6: **RAISE** (Validity Violation) {Aperiodic Consumer is still called. But it knows data is invalid}
7: **end if**
8: Set CurrentData ← data
9: APEX::Start(AperiodicConsumer Process) {Process management call}
10: **end loop**
End Aperiodic Consumer Trigger Process

Given: Name {Name of the consumer}
Given: CurrentData { Current Data variable filled in by the trigger.}
Given: COMP {implementation class for the component}
Begin Aperiodic Consumer {Partition Scheduler starts the deadline violation monitor}
1: COMP→writelock($DeadlineTime$,$return\_code$)
2: **if** $return\_code$ is TIMEOUT **then**
3: **RAISE** (LOCK TIMEOUT ERROR)
4: **return**
5: **end if**
6: Read CurrentData
7: Check pre-conditions
8: **if** pre-conditions not met **then**
9: **RAISE** (pre-condition Violation)

10: **return**
11: **end if**
12: USER_HANDLE_CONSUMER_NAME_DATA(data) {User function details are filled in by the system developer. Name is replaced by the actual name upon generation}
13: **if** user exception occurred **then**
14: **RAISE** (user exception)
15: **return**
16: **end if**
17: Check Post-conditions
18: **if** Post-conditions not met **then**
19: **RAISE** (Post-conditions Violation)
20: **return**
21: **end if**
22: COMP→unlock()
23: **return**
End Aperiodic Consumer {Partition Scheduler stops the deadline violation monitor}

B.4 Code Template for provided interface methods and associated server side skeleton

Each method in the interface supported by the Synchronous ports (provides and requires) has its separate ARINC process. This is required because each method inside an interface can have different real time properties. Following paragraph describes the template that is used to generate the code for a provided interface method. We implement the skeleton generated by the MICO CCM IDL compiler for the provided method as shown in the first part. This function runs inside an ORB worker thread, which has infinite time deadline. Upon receipt of a service call, the corresponding ARINC process is started. Note that the ARINC services prevent more than one concurrent service calls to a provided method by ensuring that a process cannot be started unless it is in the ready state i.e. not already started [1].

Given:
Begin Implementation of skeleton of the provided interface method. {This provides implementation for the provided method}
1: Store input data in shared variable
2: APEX::Start(Provided interface method ARINC process) {This is one of the process management calls implemented by the ARINC Emulation Layer}
3: Retrieve result from the shared variable
End Server side skeleton

Given: Interface Name {Name of the provided interface}
Given: Method Name {Name of the provided method in the interface}
Given: CurrentData { Current Data variable filled in by the trigger.}
Given: COMP {implementation class for the component}
Begin Provided Interface Method {Partition Scheduler starts the deadline violation monitor}
1: COMP→writelock($DeadlineTime$,$return\_code$)
2: **if** $return\_code$ is TIMEOUT **then**
3: **RAISE** (LOCK TIMEOUT ERROR)
4: **return**
5: **end if**
6: Read CurrentData
7: Check pre-conditions
8: **if** pre-conditions not met **then**
9: **RAISE** (pre-condition Violation)
10: **return**
11: **end if**
12: USER_HANDLE_PROVIDED_INTERFACENAME_METHODNAME(data) {User function details are filled in

by the system developer. The Interface name and method name are tokens and are replaced with real values upon generation.}

13: **if** user exception occurred **then**
14: **RAISE** (user exception)
15: **return**
16: **end if**
17: Check Post-conditions
18: **if** Post-conditions not met **then**
19: **RAISE** (Post-conditions Violation)
20: **return**
21: **end if**
22: COMP→unlock()
23: **return**

End Provided Interface Method {Partition Scheduler stops the deadline violation monitor}