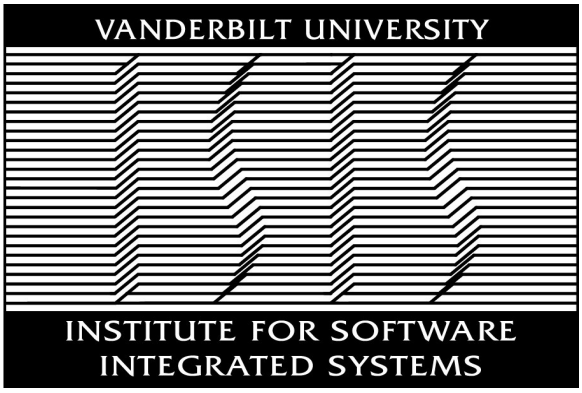

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37212

Formalization of a Component Model for Real-time Systems

Abhishek Dubey, Gabor Karsai, Nagabhushan Mahadevan

TECHNICAL REPORT

ISIS-12-102

April, 2012

Formalization of a Component Model for Real-time Systems

Abhishek Dubey, Gabor Karsai, and Nagabhushan Mahadevan

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37212, USA

Abstract. Component-based software development for real-time systems necessitates a well-defined 'component model' that allows compositional analysis and reasoning about systems. Such a model defines what a component is, how it works, and how it interacts with other components. It is especially important for real-time systems to have such a component model, as many problems in these systems arise from poorly understood and analyzed component interactions. In this paper we describe a component model for hard real-time systems that relies on the services of an ARINC-653 compliant real-time operating system platform. The model provides high-level abstractions of component interactions, both for the synchronous and asynchronous case. We present a formalization of the component model in the form of timed transition traces. Such formalization is necessary to be able to derive interesting system level properties such as fault propagation graphs from models of component assemblies. We provide a brief discussion about such system level fault propagation templates for this component model.

1 Introduction

Component-based software development is based on the notion that software should not be designed and built as monolithic artifact, rather, it should be assembled from pre-fabricated and pre-tested components, which encapsulate separable parts of a software system that implement a specific service or a set of services. A *component model* is a conceptual tool: it defines what a software component is, how it can be customized, how it can be deployed, how it is executed, and how the components interact with each other. Typically the component model also specifies a *component platform*; a middleware software layer that implements all the services needed for deploying and executing components, and for the component interactions. The primary goal of the component-based approach is to promote reusability, to increase the productivity of developers, and to ensure that the expectations from the system can be mapped to the capabilities of the components. The component model effectively establishes a contract between the components and the underlying low-level software platform.

Even though the concept of component-based software development is of the same age as software engineering[16], it has not reached the degree of practicality and popularity one finds in other fields of engineering (e.g. digital system design). One reason could be that actually there are not too many standardized ways of defining software components (c.f. the digital circuit building blocks: standard gates and storage elements) that would allow compositional reasoning. In the commercial world Microsoft's COM (and its follower: .NET), Java's Enterprise Beans, and CORBA's Component Model (CCM), are, arguably, the main examples. Components that are executed concurrently are especially problematic. The key to using concurrent components is the abstraction of the end-to-end communication dependencies, the interaction semantics, and the resultant behaviors. The semantics must be formally defined for the component model; along with other required properties such as resource needs and expected quality of service attributes for the system of components. The hope is that the system of components can be analyzed and the system-wide behaviors determined at design-time, based on the component model. But most commercial component models lack formally defined semantics, making formal reasoning about systems assembled from components difficult, if not impossible. The problem is especially acute in real-time embedded systems: there are few component models defined [5] [10] [1], and even fewer are defined precisely [3]. The situation is further complicated by the fact that in real-time systems physical time is a first-class entity, and it is not quite clear how computation and physical time can or shall be combined.

In this paper we present a component model for real-time systems and show its formalization. The component model has been informally described elsewhere [8], the purpose here is its formal presentation.

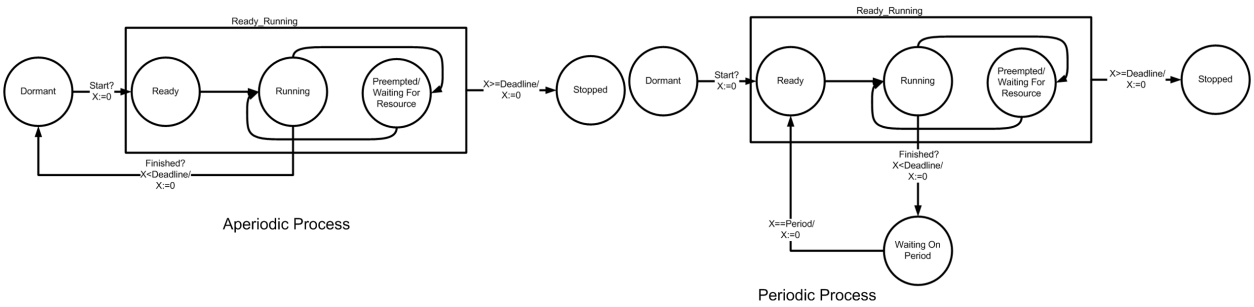

Fig. 1. States of an aperiodic process and a periodic process

The outline of the paper is as follows: Section 2 describes the ARINC-653 Component Model (ACM). Sections 3 and 4 describe the ACM semantics using Timed Transition Graphs. Section 6 presents the related research. Finally, conclusion is presented.

2 The ARINC-653 Component Model

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. ARINC-653 systems 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*. Two kinds of processes can exist inside an ARINC-653 partition: *aperiodic* and *periodic*. Periodic processes are released periodically at a specified rate. Aperiodic processes are only executed sporadically when a "start" command is issued by any other process. All processes have deadlines: during each execution, the scheduler monitors the state of the processes and raises an error if the deadline has been violated. It should be noted that careful planning is required to ensure that the processes in a partition can be scheduled. Figure 1 shows the timed automaton for an aperiodic process. It also shows the timed automaton model for a periodic process. The first start event is issued by the partition upon initialization. The scheduler is responsible for transition from ready to running state. Thereafter, any preemption or wait for resource causes the transition to the waiting state. If the finished condition is true, the process is running and the deadline has not been violated, the process goes to the waiting on period state (for periodic process) or dormant state (for aperiodic process).

The ARINC-653 component model (ACM) allows the developers to group a number of ARINC-653 processes into a reusable component. Each component can have four ports, **publishers, subscribers, facets** (provided interfaces - An interface is a collection of related methods.), and **receptacles** (required interfaces). The business logic associated with each port is executed on a separate ARINC-653 process.

A **publisher** port is a source of events: this port is used to produce events that will be consumed by other components. Each port is associated with some activity inside of the component. A publisher port needs to be triggered to publish an event (probably after reading from some internal state variable or a hardware device). This triggering can be either periodic or aperiodic (sporadic).

A **subscriber** port, as the name suggests, acts as a sink for events published by other components. Like a publisher port, it can be triggered periodically (automatically) or aperiodically (by the arrival of an event) and it consumes an event. While an aperiodic subscriber consumes all the events published by its publisher on a FIFO basis (destructive read), a periodic subscriber samples the events published at a specified rate (nondestructive read).

A **provided interface** port (or facet) exposes the implementation of the methods defined in an interface definition. When a component has a provided interface of a specific type, we say that the component implements that interface. Note that a component can implement multiple interfaces, on different ports. While a provided interface port provides an entry point into the component, a **required interface** port is used to call out from a component. Required and provided interface ports (of different components) are connected when the component-based system is configured. We say that requests for the services of a provided interface port are coming from the required interface ports of client components. The incoming client requests are queued by the middleware and are serviced by the provided port's implementation in FIFO order.

All ports can be further qualified by attributes such as **period**, **deadline**, **pre -conditions**, and **post-conditions**. Processes associated with all types of ports (publisher, subscriber, facet, and receptacle) have to finish their operations 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 that is logged by the middleware. Note that $period \geq deadline$ is required for a HARD process.

A component can also have internal periodic methods called triggers. Triggers are automatically and periodically activated by the middleware, and they must have a finite, non-zero period. They are used to periodically verify/update the component's state.

Correctness contracts: In addition to deadlines, all ports have another set of properties that must be respected: *contracts*. These contracts are expressed as pre-conditions and post-conditions on interface methods, publishers, and subscribers. Any contract violation results in a run-time error. These conditions are typically specified over the current value, or the *change* in the value (delta) since the last invocation, or the *rate of change* of some parameters. These parameters can be (a) the event data of asynchronous calls, or (b) function-parameters of synchronous calls, or (c) 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 valid after the execution of the call. This type of reasoning is critical in achieving modular certification of software components [17].

2.1 Component interactions

Two kinds of interactions: (1) asynchronous, and (2) synchronous interactions are possible between components. The possible combination of these interactions with periodic and aperiodic triggering of processes that are bound to the respective ports gives rise to a richer set of behaviors compared to CCM.

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

Synchronous Interactions: This interaction implies call-return semantics: the caller component 'calls out' via the required interface port to the connected provided interface port of the called component. A required interface port can be associated with a provided interface port of an identical interface type. A provided port can be associated with one or more required 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 called).

Constraints on Component Interactions: Safe component interactions must satisfy the following constraints: (a) The deadline of a required port must be greater than or equal to the deadline of the interacting provided port. (b) The validity period of a subscriber must be greater than the periodicity of the publisher. Otherwise, a subscriber 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 precondition checked by the interacting destination component. If this is not true, the source component might send data that is locally valid but will violate precondition of the destination component.

Deployment: The component model also provides guidance on the allocation of tasks to a component. More than one component can be allocated to a partition. however, a component cannot span the partition boundary. Please see [8] for a detailed description. The runtime framework ensures temporal partitioning between components allocated to two different partitions.

3 Port Execution Semantics

Figure 2 shows the sequence of activities related to a component port that occur when associated ARINC-653 process is in the `Running` state. The activity `Finish` refers to the last activity of an ARINC-653 process in a `Running` state just before its transition out of the `Running` state. Figure 2 also shows the sequence of activities of message transmission. We assume the existence of a globally synchronized hyper period across all modules of the software assembly and the existence of a global clock (T).

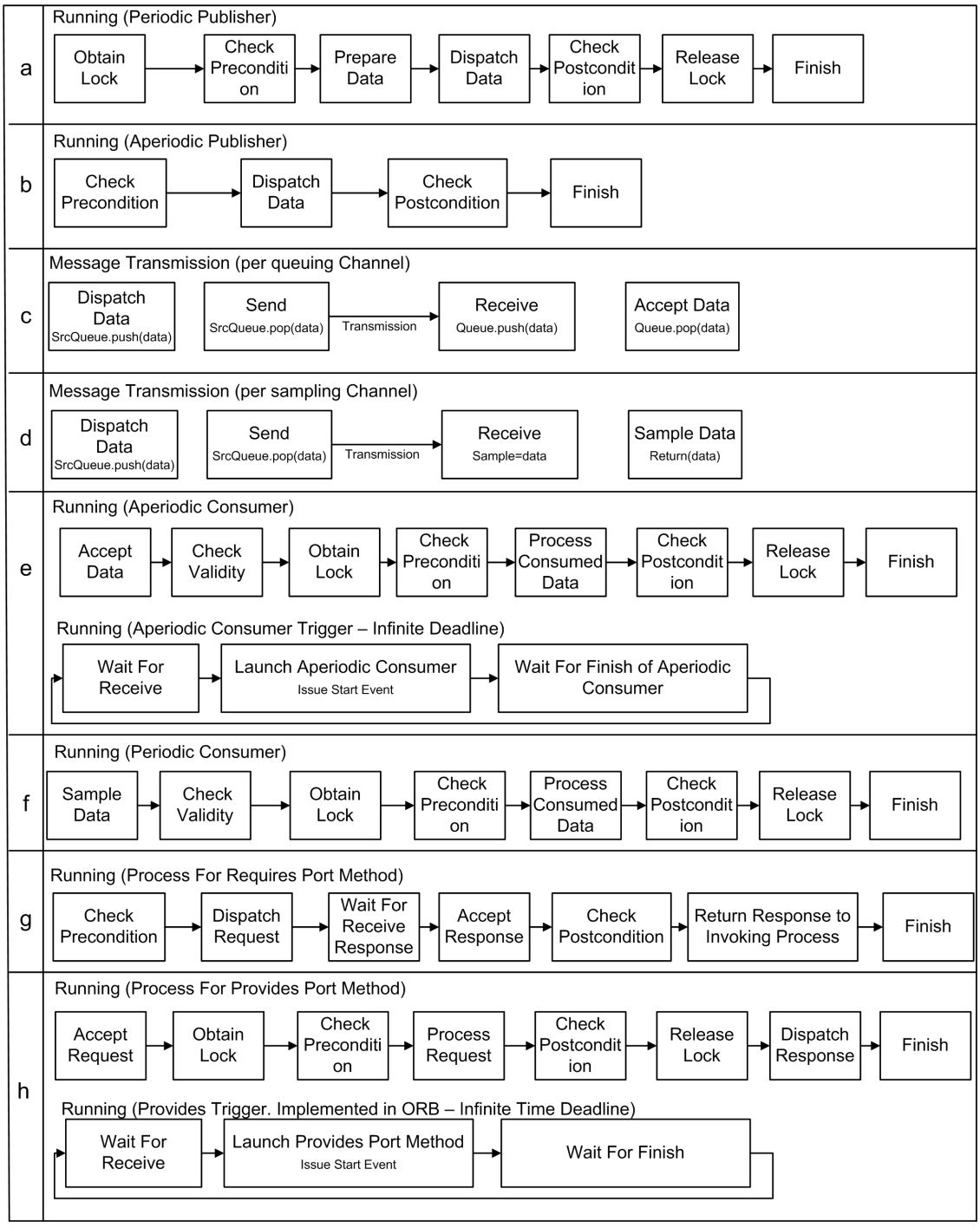

Fig. 2. Sequence of activities in the Running state of processes and channels.

In this and the next two sections, we describe the *nominal* behavior of component port and the semantics of various port to port interactions using templatized timed transition traces. Each trace is identified by a pair of values: the event/activity associated with the timed trace, and the time of occurrence of the event/activity. Additional constraints may be specified to bind the event/activity time. Unless stated otherwise, all times, denoted as $T^{ProcessType}_{EventType}(ExecutionNumber, GlobalMajorFrameNumber)$, are measured with respect to the global clock (T) that measures time relative to the start of the first global major frame.

`Event Type` refers to the completion of an activity for a particular `Process Type` as shown in Figure 2. `Execution Number` is the counter for the number of times the process has executed. `Global Major Frame Number` is the counter for the number of times the global hyper period has elapsed. For brevity, unless we specifically refer to time evaluation across two different global major frames, we will drop the second functional parameter `Global Major Frame Number` in the notation. In some cases, for example those related to message transmission, where the process execution number is irrelevant it will be dropped and marked with a $-$.

Trigger Interaction Pattern: Before we start the discussion on the semantics of ports and their interactions it is essential to describe a basic interaction pattern that occurs when one ARINC-653 process invokes another ARINC-653 process. It should be noted that only aperiodic processes can be invoked in this manner. The invoking process is called the trigger process. This pattern will be seen when we discuss the execution semantics of aperiodic subscribers and provided ports. The trigger process starts the invoked process by using an API that generates the corresponding `Start` event. Then, the trigger process enters the `waiting for resource` state. Here `resource` is a semaphore which is set when the invoked process finishes. Any processes interacting in this manner must satisfy: $Deadline^{Trigger} \geq Deadline^{InvokedProcess}$. The global hyper period is the least common multiple of the period of all processes and partitions in an ARINC-653 module. The constant HP in following sections is the global hyper period.

Periodic Publisher (PP): Execution of a periodic publisher starts when its ARINC-653 process transitions to the `ready` state (Fig. 1), which happens every $Period^{PP}$ time units. Here, $Period$ denotes the time period of the periodic publisher. The first execution of the periodic process is delayed by a phase shift, ϕ^{PP}, which is the time when the `Start` was first called for the process. Let K be the corresponding global major frame number $T^{PP}_{Ready}(N,K) = max\{(N-1),0\} * Period^{PP} + \phi^{PP}$. The global major frame counter, K, and the process execution counter, N, should satisfy the following relationship: $K * HP + \phi^{PP}\rangle = T^{PP}_{Ready}(N,K)\rangle(K-1) * HP + \phi^{PP}$. As mentioned earlier, we will drop the index K, unless we consider execution runs across two different major frames.

Once the publisher enters the running state, it attempts to obtain the lock on the component. Then, it checks the pre-conditions, passes control to the user level code to prepare the data, and dispatches the data by placing it into a queue. The data transfer happens asynchronously, Figures 2 c and d). Then it checks the post-conditions, releases the lock and finishes execution. Any failure while obtaining lock, checking pre-conditions, preparing data (user code), and checking post conditions interrupts the nominal execution and reports the discrepancy to the component health manager that handles the anomaly. The timed transition trace for nominal execution for periodic publisher is as follows:

$\langle Ready, T^{PP}_{Ready}(N)\rangle, \langle Lock, T^{PP}_{Lock}(N)\rangle, \langle PreCond, T^{PP}_{Pre}(N)\rangle, \langle Prepare, T^{PP}_{Prepare}(N)\rangle, \langle Dispatch, T^{PP}_{Dispatch}(N)\rangle,$
$\langle PostCond, T^{PP}_{Post}(N)\rangle, \langle ReleaseLock, T^{PP}_{Release}(N)\rangle, \langle Finish, T^{PP}_{Finish}(N)\rangle.$

The execution must satisfy the following condition:$T^{PP}_{Finish}(N) - T^{PP}_{Ready}(N) \leq Deadline^{PP}$

Aperiodic Publisher (AP): An aperiodic publisher in invoked by some other component process, which generates the `Start` event causing the aperiodic process associated with the AP to transition to ready state. This is the start of execution as shown in figure 1. Upon invocation, the invoking process enters the `Waiting For Resource` state. It goes back to the running state when the invoked aperiodic publisher finishes execution. Given that the invoking process belongs to the same component and is still in `Running` state, unlike a periodic publisher, the aperiodic publisher does not need to obtain the component lock. Moreover, the data to be published is passed from the invoking process to the AP and hence it does not need to spend any time in the preparing data stage. The rest of its activities are similar to periodic publisher. The timed transition trace for nominal execution is given as:

$\langle Ready, T^{AP}_{Ready}(N)\rangle, \langle PreCond, T^{AP}_{Pre}(N)\rangle, \langle Dispatch, T^{AP}_{Dispatch}\rangle, \langle PostCond, T^{AP}_{Post}(N)\rangle , \langle Finish, T^{AP}_{Finish}(N)\rangle,$

subject to $T^{AP}_{Finish}(N) - T^{AP}_{Ready}(N) \leq Deadline^{AP}$.

Message Transmission (MT): Ports of components residing in different partitions are connected via channels. These channels can connect one source port to multiple destination ports and are responsible for transmitting the messages. A channel transmitting messages across modules can be implemented as a virtual link in Avionics Full Duplex Switched Ethernet Network (AFDX) [2]. Figures 2 c) and d) show the sequence of activities that happens when a message is dispatched from one of the ports. The act of dispatching a message just puts it into a network queue. Transmission channel `sends` the message over the network when the prescribed time comes. Upon reaching the recipient, the message is `received` and pushed into the application queue, if the receiving port has a queue. Otherwise, the new data overwrites the old data in a sampling port. The timed transition trace for message transmission is given as:

$\langle Dispatch, T^{*}_{Dispatch}(-, J)\rangle$, $\langle Send, T^{MT}_{Send}(-, K)\rangle$, $\langle Receive, T^{MT}_{Receive}(-, L)\rangle$,
where $L \geq K \geq J$ and $L - J \leq 1$.

That is, at most one global hyper period can pass between the dispatch of a message and when it is actually sent over the network by the channel. Values of T^{MT}_{Send} can be pre-specified for each ARINC-653 module using a time-triggered schedule [12].

Periodic Subscriber (PS): A periodic subscriber samples the data it receives via the channel periodically. Like a periodic publisher, the execution of periodic subscriber starts when its ARINC process enters the ready state. The first execution of the periodic process is delayed by a phase shift, ϕ^{PS}, which is the time when `Start` was called for the process. The time when execution of periodic subscriber starts for the N^{th} time in the K^{th} run of the hyper period can be written as $T^{PS}_{Ready}(N, K) = max\{(N-1), 0\} * Period^{PS} + \phi^{PS}$. Once the PS enters the `running` state, it checks the validity (freshness) of the sampled data. Then it attempts to obtain the lock on the component, checks the preconditions, passes control to user level code to process consumed data, checks post conditions, releases lock and finishes execution. Any failure in these stages interrupts current execution trace. The timed transition trace for the N^{th} nominal execution for periodic subscriber is:

$\langle Ready, T^{PS}_{Ready}(N)\rangle$, $\langle Sample, T^{PS}_{Sample}(N)\rangle$, $\langle Validity, T^{PS}_{Validity}(N)\rangle$, $\langle Lock, T^{PS}_{Lock}(N)\rangle$, $\langle PreCond, T^{PS}_{Pre}(N)\rangle$,
$\langle Process, T^{PS}_{Process}(N)\rangle$, $\langle PostCond, T^{PS}_{Post}(N)\rangle$, $\langle ReleaseLock, T^{PS}_{Release}(N)\rangle$, $\langle Finish, T^{PS}_{Finish}(N)\rangle$,

subject to $T^{PS}_{Finish}(N) - T^{PS}_{Ready}(N) \leq Deadline^{PS}$.

Aperiodic Subscriber (AS) Figure 2 (e) describes the various stages that happen when the aperiodic subscriber is running. A separate process called aperiodic subscriber trigger is responsible for releasing the aperiodic subscriber upon receipt of an incoming data. Other difference between periodic subscriber and aperiodic subscriber is that while the *sample data* activity of periodic subscriber is a non-destructive read i.e. the same data item can be read again, the *accept data* activity is a destructive read. Apart from that all other activities of aperiodic subscriber are similar to periodic subscriber. The timed transition trace for the N^{th} nominal execution for aperiodic subscriber:

$\langle Ready, T^{AS}_{Ready}(N)\rangle$, $\langle Accept, T^{AS}_{Accept}(N)\rangle$, $\langle Validity, T^{AS}_{Validity}(N)\rangle$, $\langle Lock, T^{AS}_{Lock}(N)\rangle$, $\langle PreCond, T^{AS}_{Pre}(N)\rangle$
, $\langle Process, T^{AS}_{Process}(N)\rangle$, $\langle PostCond, T^{AS}_{Post}(N)\rangle$, $\langle ReleaseLock, T^{AS}_{Release}(N)\rangle$, $\langle Finish, T^{AS}_{Finish}(N)\rangle$,

subject to $T^{AS}_{Finish}(N) - T^{AS}_{Ready}(N) \leq Deadline^{AS}$.

Provided Port (PrP): A provided interface port contains the implementation for the methods defined in the provided interface and services the request issued on these interfaces from a connected required port. Their interaction implies call-return semantics. Both the provided and required ports are always aperiodic. The behavior of provided port is similar to the behavior of an aperiodic subscriber. Here, again, a trigger process (mentioned earlier in the section) is responsible for invoking the provided port process by issuing the `Start` event when a request is received. If multiple requests for the provided port exist, they are queued by at the receiving end and serviced in FIFO. Figure 2 (h) shows the sequence of activities in the `running` state

of a provided port process. Key differences between this port and aperiodic subscriber are that provided port process does not check the age of incoming data and that it also acts like an aperiodic publisher and dispatch the response to be transmitted by the communication channel. The timed transition trace for this port can be written as:

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

subject to $T^{PrP}_{Finish}(N) - T^{PrP}_{Ready}(N) \leq Deadline^{PrP}$.

Required Port (RP): Like aperiodic publishers, a process for required port is set to ready state when some other process in the component invokes the corresponding method on the required port. Figure 2 (g) shows sequence of activities that occur when the required process is in the `running` state. Note that similarly to the aperiodic publisher, the required port process does not need to obtain a component lock as the invoking process already holds that lock and is blocked till the invoked process is finished. The key difference between aperiodic publisher and the required port is that after dispatching the request, the required port process blocks itself and waits to receive the corresponding response from a provides port. Once it receives the response, it accepts it, checks the post conditions, and returns the response via shared memory to the invoking process and finishes. Note that unlike the interaction between asynchronous ports, communication over synchronous ports results in two pairs of dispatches: send and receive over two different communication channels[1]. The timed transition trace for nominal execution is given as:

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

The execution must satisfy the following condition $T^{RP}_{Finish}(N) - T^{RP}_{Ready}(N) \leq Deadline^{RP}$. Clearly the delay from dispatching the request, transmission of message across the network, processing of request on the provides port, and transmission of response back need to be accounted for while setting the deadline of the required port.

4 Interaction Semantics

Asynchronous Interactions: These interactions occur when a publish port of a component is connected to a subscriber port of another component. While a subscriber can be connected to only one publisher, a publisher may be connected to one or more subscribers. Strict type matching on the event type is required between the publisher and its subscribers. Note that the semantics of interaction between a publisher and subscriber are governed by the nature of the subscriber.

Interaction between a Periodic Subscriber and a Publisher A periodic subscriber 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 subscriber ensures that the events from the publisher are sampled at a specific rate. When both the interacting publisher and subscriber are periodic, the value of the subscriber's period relative to the publisher's determines if the subscriber 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). The data dispatched by a publisher is a asynchronously consumed by the periodic subscriber. The timed trace for the interaction from the point of dispatch to the point of data validity check can be written as follows:

$\langle Dispatch, T^{*}_{Dispatch}(-, K_0)\rangle$, $\langle Send, T^{MT}_{Send}(-, K_1)\rangle$, $\langle Receive, T^{MT}_{Receive}(-, K_2)\rangle$,

[1] A communication channel in ARINC-653, can only have one source, but can have multiple destinations

$\langle Ready, T^{PS}_{Ready}(-, K_3)\rangle$, $\langle Sample, T^{PS}_{Sample}(-, K_4)\rangle$, $\langle Validity, T^{PS}_{Validity}(-.K_5)\rangle$.

Here $*$ means either AP or PP. $-$ for the first index implies that we are referring to the latest execution of the process in the specified run of the hyperperiod. Also,

$K_5 - K_0 \leq 1$, $K_5 \geq K_4 \geq K_3 \geq K_2 \geq K_1 \geq K_0$,and $T^{PS}_{Validity}(-, K_5) - T^{*}_{Dispatch}(-, K_0) \leq Validity$,

where validity is the maximum staleness of data specified at the subscriber. In the case of the periodic subscriber, once the data is received on the subscriber sampling port, the data is consumed at the next invocation of the subscriber process i.e. there can be at most one hyperperiod difference between dispatch and sampling of the data at the subscriber end. Additionally, for a nominal execution the sampled data's age should be less than the specified validity value.

Interaction between an Aperiodic Subscriber and a Publisher: Interaction between a publisher and an aperiodic subscriber is indicative of a pattern where the subscriber is reactive in nature. In such a case, the subscriber 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 subscriber to operate on all of the events received. The data dispatched by a publisher is a asynchronously consumed by the connected aperiodic subscriber. The timed trace for the interaction from the point of dispatch to the point of data validity check can be written as follows:

$\langle Dispatch, T^{*}_{Dispatch}(-, K_0)\rangle$, $\langle Send, T^{MT}_{Send}(-, K_1)\rangle$, $\langle Receive, T^{MT}_{Receive}(-, K_2)\rangle$, $\langle Ready, T^{AS}_{Ready}(-, L_0)\rangle$,
$\langle Accept, T^{PS}_{Sample}(-, L_1)\rangle$, $\langle Validity, T^{AS}_{Validity}(-, L_2)\rangle$.

Here $*$ means either AP or PP. $-$ for the first index implies that we are referring to the latest execution of the process in the specified run of the hyperperiod. Also,
$K_2 - K_0 \leq 1$, $L_2 \geq L_1 \geq L_0 \geq K_2 \geq K_1 \geq K_0$, $T^{AS}_{Validity}(-, L_2) - T^{*}_{Dispatch}(-, K_0) \leq Validity$.

It should be noted here that there is a time difference of at most one hyperperiod between dispatch from the publisher and receipt of the data at the subscriber queue. However, since the data to be consumed is queued, it is quite possible that the aperiodic subscriber may get to the data only in a future frame. Of course, for a nominal operation it is desired that at the point of consumption the sampled data's age should be less than the specified validity value.

Synchronous Component Interactions: These interactions imply 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 required 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 called). Synchronous ports in this model are always aperiodic. The interaction patterns observed in synchronous ports is borrowed from CCM. The key difference is deadline monitoring. The default type of interaction is call-return or two-way communication i.e. the Required port waits for the provided port to finish its operation and return the results. A one way interaction is also available. However, its semantic is similar to aperiodic publisher and subscriber with a difference that while publisher does not fail if a subscriber 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. The full interaction for a two way call can be written as follows:

$\langle Ready, T^{RP}_{Ready}(n, K)\rangle$, $\langle PreCond, T^{RP}_{Pre}(n, K)\rangle$, $\langle DispatchReq, T^{RP}_{DispatchReq}(N, K)\rangle$,
$\langle Send, T^{MT1}_{Send}(-, L)\rangle$, $\langle Receive, T^{MT1}_{Receive}(-, M)\rangle$,

$\langle Ready, T^{PrP}_{Ready}(m, M)\rangle$, $\langle Accept, T^{PrP}_{Accept}(m, M)\rangle$, $\langle Lock, T^{PrP}_{Lock}(m, M)\rangle$,
$\langle PreCond, T^{PrP}_{Pre}(m, M)\rangle$, $\langle Process, T^{PrP}_{Process}(m, M)\rangle$, $\langle PostCond, T^{PrP}_{Post}(m, M)\rangle$,
$\langle ReleaseLock, T^{PrP}_{Release}(m, M)\rangle$, $\langle DispatchResponse, T^{PrP}_{DispatchRes}(m, M)\rangle$, $\langle Send, T^{MT2}_{Send}(-, N)\rangle$,

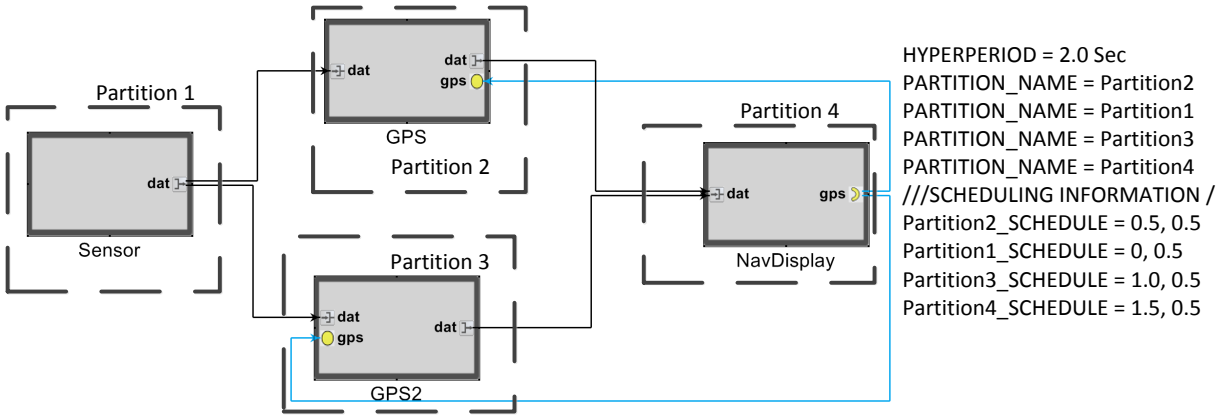

Fig. 3. GPS Software Assembly used in the case study - Unit of time is seconds.

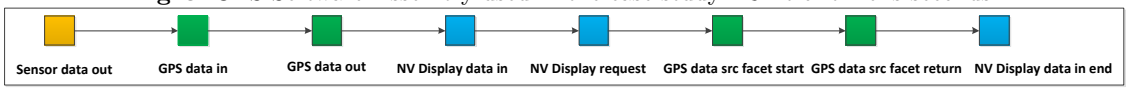

Fig. 4. Chain of Events associated with data production and consumption in a hyper period.

$\langle Receive, T^{MT2}_{Receive}(-,O)\rangle$, $\langle Accept,\ T^{RP}_{AcceptRes}(n,P)\rangle$, $\langle PostCond, T^{RP}_{Post}(n,P)\rangle$, $\langle ReturnResponse, T^{RC}_{RR}(n,P)\rangle$, $\langle Finish, T^{RP}_{Finish}(n,P)\rangle$.

This timed trace shows the n^{th} execution of the required port and the m^{th} execution of the provided port. K, L, M, N, O, P are global major frame instances such that $K \leq L \leq M \leq N \leq O \leq P$. And $P - K \leq 1$, that is the full interaction between these ports should finish maximum within current and the next global major frame. That is:

$T^{RP}_{Ready}(n,K) \leq T^{MT1}_{Send}(-,L) \leq T^{MT1}_{Receive}(-,M) \leq T^{MT2}_{Send}(-,N) \leq T^{MT2}_{Receive}(-,O), T^{RP}_{Finish}(n,P) - T^{RP}_{Ready}(n,K) \leq Deadline^{RP}$, and $T^{PrP}_{Ready}(m,M) - T^{PrP}_{DispatchRes}(m.M) \leq Deadline^{PrP}$

The schedule for message transmission on these channels should be such that the cumulative delay encountered in both message transfers plus the execution time of the called method is still less than the deadline of the required port. Moreover, the deadline of the required port must be less than the deadline of the process that invoked it, which is required by the constraint of the trigger pattern, as described at the start of section 3.

Example: Figure 4 shows the connections between the components and their deployment on four different partitions. Partition 1 contains the Sensor Component. Partition 2 contains the GPS and Partition 4 contains the Navigation Display component. The sensor component publishes an event every 4 sec. The GPS component consumes the event published by sensor at a periodic rate of 4 sec. Afterwards it publishes an event, which is sporadically consumed by the Navigation Display (abbreviated as display). Thereafter, the display component updates its location by using getGPSData facet of the GPS Component. The publisher-subscriber interaction between sensor and GPS components is implemented via a sampling port (Sampling ports are basic inter-partition communication mechanism in ARINC 653 platforms). A Channel connects the source sampling port from partition 1 to destination sampling port in partition 2. In this example, a redundant GPS is also connected in the assembly. The redundant component in this case shares the port structure with the other GPS. However, their internal behaviors are different. In this particular example, GPS 2 is set to the semi-active State i.e. it can consume but not publish.

Figure 4 also describes the periodic schedule followed by the partitions, overseen by a controller process called Module Manager [6]. This schedule is repeated every 2 s (hyper period). In each cycle, Partition 1 runs with a phase of 0 sec for 500 ms (duration). Partition 2's phase is 500 ms. It runs for 500 ms (duration).

Then Partition 3 and Partition 4 run for next 1 second. This ensures that the two partitions are temporally isolated. A nominal execution sequence is shown in figure 4.

5 Interaction Semantics and Fault Propagation Graphs

Information present in the ACM assembly model, combined with the knowledge of the port-execution semantics and the port-interaction semantics described in the previous sections is used to automate the synthesis of fault-propagation models, within and across ports, within and across components in the assembly. This approach is similar to the failure propagation and transformation calculus described by Wallace [21].

The fault propagation model used in our study is based on the concept of Timed Failure Propagation Models described in [11,14]. This TFPG model is built as a hierarchy: starting with the TFPG models for the component ports, using the component port TFPG models along with the data and control flow information (captured in these models) to build the TFPG models of the Components, and using the Component TFPG models and the information on the port-interactions across component boundaries to build the TFPG model of the entire Assembly. Here we provide a brief discussion highlighting the relationship between the port-execution and port-interaction semantics and the TFPG models. For a detailed discussion readers should refer to [7].

Figure 5 shows a portion of the TFPG model of the entire GPS Assembly (figure 4). In this discussion, we restrict ourselves to kinds of root failure causes (a) a latent bug in the implementation e.g. FM_USER_CODE in fig 5 , (b) problem introduced in the operating environment - e.g. problems in synchronization: FM_ LOCK_PROBLEM in fig 5. Effect of these faults manifest as a violation in one or more of the executed operations. These anomalies or discrepancies include those related to securing the lock (DISC_LOCK_TIMEOUT_FAILURE), validity check (DISC_VALIDITY_FAILURE), pre-condition violation (DISC_PRECONDITION_FAILURE), user-code exception (DISC _USER_CODE), post-condition violation (DISC_POST_CONDITION), deadline violation (DISC _DEADLINE). Additionally, these primary anomalies could be affected by input discrepancies, e.g. problems caused by a component providing the data (e.g. . DISC_BAD_DATA_IN) that cause the cascading effect of failures from outside.

Fault Cascades: The failure effect from the port operation cascade out of the port through the output discrepancies (e.g. DISC INVALID/ LATE/ NO DATA PUBLISHED in publisher port of the Sensor and DISC_PROBLEM_WITH_STATE_UPDATE in GPS's subscriber port). Additional discrepancies or anomalies considered in the template TFPG models include those related to bad-values in the state-variable of the components (e.g. DISC_SensorValue_Bad in Sensor , GPSValue_Bad_State in GPS), and those related to failure effects that affect invocation of aperiodic ACM-ports (e.g. DISC_Not_Invoked in GPS's data_in subscriber port). Notice that these anomalies coincide with the different states in the timed transition graph of the interaction semantics described earlier. These failure propagation links are primarily derived by the following contexts: **1)** Failure Propagation related to ACM-Port operation is determined by different discrepancies that can be detected in the different states in the port execution trace. For example, in the GPS' subscriber data_in this failure propagation path starts from the validity violation related to staleness of data, DISC_VALIDITY_FAILURE and proceeds through DISC _PRECONDITION _FAILURE, DISC_USER_CODE leading to DISC_POST_CONDITION and DISC_ DEADLINE. **2)** Failure propagation within an ACM Component is either caused by bad data (e.g. an incorrect update of the state variables) or by propagations that can be deduced from a model of the control flow within the component. **3)** Finally, cascading failure propagation across ACM components are dictated by the ACM-port interaction semantics described earlier. For example, the asynchronous interaction between a publisher and subscriber results in one-way failure propagation from the publisher to the subscriber. This is because of lack of data published, or bad data published or data published late resulting in problems in the subscriber. In the figure 5, this is evident from the failure propagation links captured from Sensor's publisher port: DISC INVALID DATA PUBLISHED, DISC LATE DATA PUBLISHED, DISC NO DATA PUBLISHED to GPS's subscriber port: DISC_VALIDITY_FAILURE, DISC_BAD_DATA_IN. Note that due to its nature, the two way synchronous call can result in failure propagation in either direction. Failure propagation resulting from synchronous interaction is not shown in figure 5.

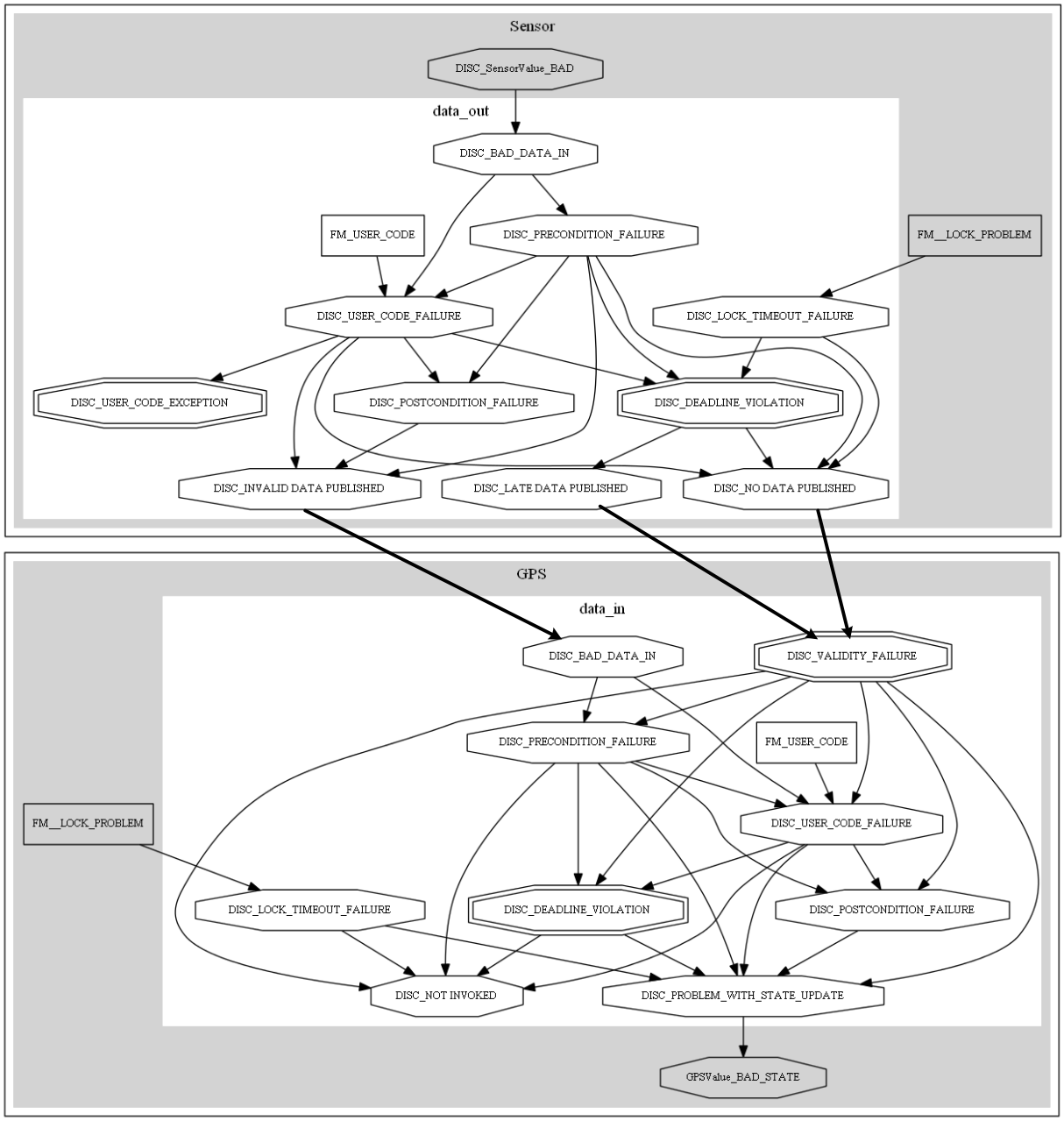

Fig. 5. TFPG model for Sensor-Publisher and GPS-subscriber

6 Related Research

TinyOS [20], ControlShell [18], eCos [9], Koala [15] 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. CIAO [4] and PRISM [19] are two component runtimes implementations 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. However there is no formal specification of the interaction semantics.

POK (PolyORB Kernel) [5] is a runtime for ARINC-653 Annex of the Architecture Analysis and Design Language (AADL) [10]. AADL is a design language for modeling the software and hardware architecture of

embedded systems the provides constructs for modeling different components in the system stack at different levels of abstraction. AADL also enables specification of interaction between components. Properties such as deadline, worst case execution time, that are critical for assessing the performance and functionality of the system can also be specified. POK uses the OCARINA[2] framework to automatically configure and deploy processes and partitions. The main difference between POK and ACM 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.

Automotive Open System Architecture (AUTOSAR) is a standardized software architecture, jointly developed by automotive manufacturers, suppliers and other key players [1]. The key motivation behind AUTOSAR is to mitigate the complexity posed by increasing software use and the business need for increased scalability and flexibility to develop and reuse software functions across different product lines. Unfortunately, the initial specification did not define a component model or interaction semantics for components.

BIP [3] defines an abstract language to specify systems as a composition of behaviors (Petri nets extended with C), interactions (rendezvous and broadcast), and priorities (that resolve scheduling conflicts). BIP supports the component-oriented modeling and construction of embedded systems, and it provides a machinery with which any component interaction semantics can be constructed.

7 Conclusions

We have briefly introduced a component model for real-time systems where components are executed concurrently and interact via well-defined interactions. We have specified the execution semantics of the components and component interactions via timed transition traces, and presented what constraints the executions must satisfy. Such formal presentation helps researchers and users of the component model in understanding the behavior exhibited by a system. Further research will look into how this formalization can be used to construct design-time analysis tools to verify the component assemblies that represent entire system. In a related project [13] we have investigated how run-time monitoring tools can be constructed based on this semantics that assist in implementing a 'software health management' function for component-based application.

8 Acknowledgments

This report is based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. The authors would like to thank Paul Miner, Eric Cooper, and Suzette Person of NASA Langley Research Center for their help and guidance on the project.

References

1. AUTOSAR technical overview. Tech. rep., http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf
2. Alena, R.L., Ossenfort, J.P., Laws, K.I., Goforth, A., Figueroa, F.: Communications for integrated modular avionics. In: Proc. IEEE Aerospace Conference. pp. 1–18 (3–10 March 2007), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4161517
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods. pp. 3–12. IEEE Computer Society, Washington, DC, USA (2006), http://dl.acm.org/citation.cfm?id=1158333.1158344
4. CIAO. http://download.dre.vanderbilt.edu/
5. Delange, J., Pautet, L., Feiler, P.: Validating safety and security requirements for partitioned architectures. In: Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies. pp. 30–43. Springer-Verlag, Berlin, Heidelberg (June 2009)

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

6. Dubey, A., Karsai, G., Kereskenyi, R., Mahadevan, N.: A Real-Time Component Framework: Experience with CCM and ARINC-653. Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on pp. 143–150 (2010)
7. Dubey, A., Karsai, G., Mahadevan, N.: Towards model-based software health management for real-time systems. Tech. Rep. ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University (August 2010), http://isis.vanderbilt.edu/node/4196
8. Dubey, A., Karsai, G., Mahadevan, N.: A component model for hard real-time systems: CCM with ARINC-653. Software: Practice and Experience 41(12), 1517–1550 (2011), http://dx.doi.org/10.1002/spe.1083
9. eCos, http://ecos.sourceware.org/
10. Feiler, P., Lewis, B., Vestal, S., Colbert, E.: 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, IFIP International Federation for Information Processing, vol. 176, pp. 3–15 (2005)
11. Hayden, S., Oza, N., Mah, R., Mackey, R., Narasimhan, S., Karsai, G., Poll, S., Deb, S., Shirley, M.: Diagnostic technology evaluation report for on-board crew launch vehicle. Tech. rep., NASA (2006)
12. Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K.: The time-triggered ethernet (tte) design. Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on 0, 22–33 (2005)
13. Mahadevan, N., Dubey, A., Karsai, G.: Application of software health management techniques. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 1–10. SEAMS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1988008.1988010
14. Ofsthun, S.C., Abdelwahed, S.: Practical applications of timed failure propagation graphs for vehicle diagnosis. In: Proc. IEEE Autotestcon. pp. 250–259 (17–20 Sept 2007)
15. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. Computer 33(3), 78–85 (2000)
16. Peter, N., Randell, B.: Software engineering: Report of a conference sponsored by the nato science committee (711 october 1968), garmisch, germany. Scientific Affairs Division, NATO (1968), http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
17. Rushby, J.: Modular certification. Tech. rep. (Sep 2001)
18. Schneider, S., Chen, V., Steele, J., Pardo-Castellote, G.: The controlshell component-based real-time programming system, and its application to the marsokhod martian rover. SIGPLAN Not. 30(11), 146–155 (1995)
19. Schulte, M.: Model-based integration of reusable component-based avionics systems - a case study. In: ISORC 2005. pp. 62–71
20. Tinyos, http://webs.cs.berkeley.edu/tos/
21. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electron. Notes Theor. Comput. Sci. 141(3), 53–71 (2005)