F6COM: A Component Model for Resource-constrained and Dynamic Space-based Computing Environments

William R. Otte, Abhishek Dubey, Subhav Pradhan, Prithviraj Patil,
Aniruddha Gokhale, and Gabor Karsai*
* ISIS, Dept of EECS, Vanderbilt University,
Nashville, TN 37235, USA
Email:{wotte,dabhishe,pradhasm,prithviraj6116,gokhale,gabor}@isis.vanderbilt.edu

Johnny Willemsen†
† Remedy IT
2650 AC Berkel en Rodenrijs
The Netherlands
Email: jwillemsen@remedy.nl

***Abstract*—Component-based programming models are well-suited to the design of large-scale, distributed applications because of the ease with which distributed functionality can be developed, deployed, and validated using the models' compositional properties. Existing component models supported by standardized technologies, such as the OMG's CORBA Component Model (CCM), however, incur a number of limitations in the context of cyber physical systems (CPS) that operate in highly dynamic, resource-constrained, and uncertain environments, such as space environments, yet require multiple quality of service (QoS) assurances, such as timeliness, reliability, and security. To overcome these limitations, this paper presents the design of a novel component model called F6COM that is developed for applications operating in the context of a cluster of fractionated spacecraft. Although F6COM leverages the compositional capabilities and port abstractions of existing component models, it provides several new features. Specifically, F6COM abstracts the component operations as tasks, which are scheduled sequentially based on a specified scheduling policy. The infrastructure ensures that at any time at most one task of a component can be active — eliminating race conditions and deadlocks without requiring complicated and error-prone synchronization logic to be written by the component developer. These tasks can be initiated due to (a) interactions with other components, (b) expiration of timers, both sporadic and periodic, and (c) interactions with input/output devices. Interactions with other components are facilitated by ports. To ensure secure information flows, every port of an F6COM component is associated with a security label such that all interactions are executed within a security context. Thus, all component interactions can be subjected to Mandatory Access Control checks by a Trusted Computing Base that facilitates the interactions. Finally, F6COM provides capabilities to monitor task execution deadlines and to configure component-specific fault mitigation actions.**

***Index Terms*—component models, cyber physical systems, cluster and cloud, wireless networking, mobility.**

I. Introduction

Component-based software engineering (CBSE) [1] is based on the notion that software should be assembled from prefabricated 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, including COM and .NET by Microsoft, the CORBA Component Model (CCM) defined by OMG and implemented by many vendors, and the Enterprise Java Beans (EJB) from Sun/Oracle, just to name the three major ones. The component models define what a component is, how it can be customized, assembled to form applications, deployed, executed, and how the components interact with each other. Each component model also defines a component platform: a middleware software layer that implements common services needed by applications. With increasing time-to-market pressures that force significant reuse and the increasing scale and complexity of applications, CBSE is generally the preferred approach to developing and deploying large-scale distributed applications.

Although CBSE has traditionally been used to develop enterprise applications, a number of prior efforts [2], [3], [4], [5] have also used CBSE for real-time and embedded applications. These component models for real-time and embedded applications focus on assuring one or more of the different domain requirements, such as meeting the different non-functional properties (*e.g.*, timeliness, reliability and security), satisfying limitations on resources, and handling uncertainties in operating environments.

The work presented in this paper describes a component model called F6COM that we have developed to operate in a real-time embedded environment of fractionated spacecraft [6] (F6 stands for Future, Fast, Flexible, Fractionated, Free-Flying [7] spacecraft). The fractionated spacecraft concept helps increase mission reliability by virtue of using smaller form factor and relatively inexpensive spacecraft that form a cluster with potentially redundant capabilities. Space missions are supported by distributed software applications whose functionality is spread across the different spacecraft in the cluster. The cluster of fractionated spacecraft essentially provides a cloud computing platform [8] in space where different, potentially concurrent, missions can lease resources on the spacecraft cluster for their needs.

Although computing on fractionated spacecraft shares many of the same stringent requirements as other distributed real-time embedded systems, it also presents a unique set of challenges distinct from other such systems. For example,

it is important to assure robust and reliable operations since runtime debugging and firmware upgrades are often difficult, and physical access to the hardware is practically impossible. This requirement imposes the need for highly robust software (*e.g.*, free of race conditions and deadlocks). Second, space computing presents with numerous additional dimensions of uncertainties arising from faults in the hardware caused by, for instance, radiation effects. This requirement imposes the need for real-time anomaly detection and fault mitigation capabilities. Third, communication links in the cluster can exhibit a wide range of fluctuations in latency, reliability, and speed, depending on the position and attitude of the satellite and other environmental factors, which imposes the need for dynamic resource management.

An already formidable set of challenges manifested in space computing are further amplified by the fractionated spacecraft computing model, which implies that multiple missions can be simultaneously hosted on the platform. This necessitates the requirement for strict isolation between different yet concurrent missions, including strong security assurances. Since multiple space missions can be hosted on a fractionated spacecraft cluster simultaneously, the system must support different interaction semantics, such as synchronous remote method invocations, asynchronous messaging, and publish/subscribe — all subject to security constraints.

This paper therefore presents the design rationale and evaluation of a new component model called F6COM for fractionated spacecraft. The rest of the paper is organized as follows: Section II describes related research comparing it with F6COM; Section III describes the F6COM component model in detail explaining how it resolves the challenges described in this section and how it overcomes the limitations in prior work; Section IV provides results of empirically evaluating the F6COM capabilities; and finally Section V provides concluding remarks and alludes to future work.

II. Related Work

In [9], authors provide a detailed comparison of different component frameworks that are tailored towards real-time and embedded systems. For example, the Pervasive Component Systems (PECOS) [2], [10] project describes a component model for embedded systems and is specifically tailored to field devices. Field devices are reactive, embedded devices fitted with sensors and actuators, and are developed using the most inexpensive of hardware. They are severely constrained in the amount of RAM, CPU capacity and other resources. The key contributions of PECOS are its support for non-functional properties, such as maintaining hard real-time properties, and lifecycle activities, such as specification, composition, deployment and configuration. Moreover, components in PECOS could be active (which have their own thread of control and support long lived activities), passive (which do not own a thread of control and are scheduled by another active component), and event components (which are triggered by some events). PECOS also supplies the CoCo language used to specify components and their composition.

In many respects F6COM shares the same goals as PECOS: it is tailored to support multiple nonfunctional properties in resource-constrained and highly uncertain environments. F6COM also comes with model-driven engineering tools (not discussed in this paper) to specify, compose, deploy and configure F6 applications. However, there are key differences between F6COM and PECOS. F6COM does not make a distinction among component types; rather components can exist in different states that dictate their behavior. In addition to event-triggered behavior, F6COM also supports time-triggered actions. Moreover, F6COM provides finer granularity of control and communication semantics by providing a variety of port types that support both synchronous remote method invocations and publish/subscribe forms of communication. Additionally, F6COM support secure communications with appropriate support at the level of ports. Finally, unlike PECOS, which is tailored to support field devices that perform a limited set of functions, F6COM are developed as reusable units of functionality for a wide range of applications that can be executed in distributed, space computing environments.

The PROGRESS component model [4], [11] is a recent effort to develop a component model for real-time and embedded systems, most notably tailored towards supporting vehicular and telecommunications-based embedded environments. For applications based on PROGRESS, the component-based software development philosophy is used in all stages of application development. Support is provided for a variety of analysis including functional compliance, timing properties, and resource usage to realize extremely robust applications. The PROGRESS model supports two kinds of communications: the first for messages within the same physical host, which is supported through one subsystem of PROGRESS, while the second supports messages sent over the bus. PROGRESS also supports a two-level component model. The top level deals with the distribution, concurrency, and synchronization aspects and is used by components that are active and communicate through ports using asynchronous messaging. At the bottom layer is a low-level component model comprising components that are passive and are activated by events at the higher level.

F6COM shares many of the same goals of PROGRESS, *i.e.*, they both seek to realize robust, secure, and reliable distributed, real-time and embedded systems. Unlike PROGRESS, F6COM supports a single-layer component model design that supports various kinds of communication semantics. Security is also a key distinguishing characteristic. With respect to concurrency, although F6COM supports concurrent application threads, only one operation is allowed to execute at any given time in a particular component to eliminate any race conditions and the need for synchronization at the application level. Moreover, although currently we are developing F6COM for space-based computation, F6COM can easily be used in a variety of other domains.

The Component Integrated ACE ORB (CIAO) [3], [12] project is our own related effort on component middleware for distributed, real-time and embedded systems. CIAO is an implementation of the OMG's Lightweight CORBA Component

Model (LwCCM) specification [13]. CIAO uses the TAO [14] CORBA object request broker (ORB) as its default underlying communication middleware. With the recent standardization of connector mechanisms [15], CIAO is also able to support asynchronous messaging and the OMG Data Distribution Service (DDS) through its ports.

Although F6COM has been designed based on our experiences with CIAO, there are key differences. First, unlike CIAO, F6COM is not tightly coupled to the CORBA transport mechanism. All communications in F6COM are through the ports and use the connector technology [16] that enables F6COM to use a variety of communication mechanisms. Second, unlike CIAO, support for security is built into the port-based communication of F6COM. Third, unlike CIAO which borrows the threads of control from the underlying ORB to execute application logic and can potentially have multiple concurrent threads of execution, F6COM has its own thread of control. Moreover, for safety and deadlock free behavior, F6COM allows only one thread of control to be active at any given instant in time.

The ARINC-653 Component Model (ACM) [5] is another of our prior efforts in building component models for hard real-time systems. ACM implements a component model for the ARINC-653 standard [17] for avionics computing. The F6COM design also incorporates our experiences from ACM. In particular, we leverage the scheduling mechanism supported in ACM, the component states, ports and port types as well as the anomaly detection and fault mitigation capabilities that include deadline monitoring. However, as noted earlier, security in F6COM is not an afterthought but rather a first class entity in its design. Similarly, focus on robustness and deadlock/race condition-free behavior is a key design goal for F6COM.

In summary, F6COM represents a hybrid between ACM and CIAO leveraging the best features from these, and enhancing them to suit the highly dynamic and uncertain but resource-constrained space computing environment.

III. F6 Component Model

This section presents the F6COM component model by describing its salient features that distinguishes it from other component models and how they address key requirements of distributed, real-time applications that execute on a fractionated spacecraft. To better understand the F6COM model we first briefly describe the overall architecture of the fractionated spacecraft computing environment called the F6 Information Architecture Platform (F6 IAP).

A. Overview of the F6 Information Architecture Platform

The F6 IAP is a layered architecture [18] shown in Figure 1 that comprises a novel operating system, a middleware layer, and component-based applications. The operating system provides primitives for concurrency, synchronization, and secure information flows; it also enforces application separation and resource management policies. The middleware provides higher-level services supporting request/response and publish/subscribe interactions for distributed software. The

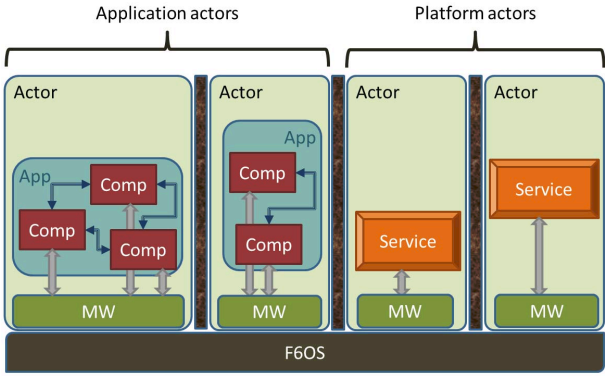

Fig. 1. F6 Information Architecture Platform

component model facilitates the creation of software applications from modular and reusable components that are deployed in the distributed system and interact only through well-defined mechanisms. Redundant copies of components can also be deployed to facilitate active fault management.

Components are grouped into *actors*: uniquely identifiable and restartable processes that are (1) temporally and spatially isolated from each other, and (2) that may be distributed and replicated across nodes. *Application actors* form applications and one application may be split across multiple application actors, potentially on different nodes and satellites. *Platform actors* provide system-level services, such as component deployment and fault management.

One of the platform actors is called the *Deployment Manager*. It provides the deployment and configuration capabilities to the system and is responsible for instantiating the components and configuring them [19]. A detailed discussion of the deployment manager is out of scope of this paper.

Two cross-cutting aspects: *multi-level security* and *multi-layered fault management* are addressed at all levels of the architecture. The complexity of creating applications and performing system integration is mitigated through the use of a domain-specific model-driven development process called Model Integrated Computing [20] that relies on a dedicated modeling language and its accompanying graphical modeling tools, software generators for synthesizing infrastructure code, and the extensive use of model-based analysis for verification and validation.

B. Design of the F6COM Component Model

We now present the F6COM component model and the rationale behind the several different design decisions we made. Figure 2 provides an overview of the various features of a F6 component. We will discuss them later in this section. The rest of this section is organized according to the decisions we made and how they relate to supporting the mission-critical applications that share the spacecraft cluster.

1) Component Lifecycle: Since components are reusable units of functionality that can be composed to create applications and subject to active fault management, individual F6COM components require a number of execution states. For example, a component that is currently being configured cannot be ready to execute the *business logic* - the non-

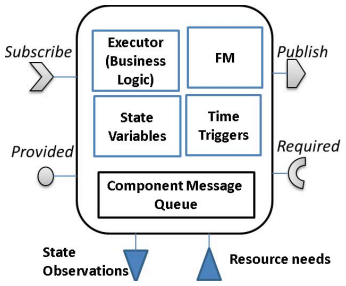

Fig. 2. F6 Component

infrastructure software provided by component developers that implements the component functionality. To address these requirements, the F6COM supports four different component execution (or lifecycle) states:

- **Initial:** This is the state in which the component starts after being instantiated. In this state the deployment infrastructure can configure the component parameters. Component parameters may only be altered in this or the inactive state, described below.
- **Passive:** In this state the component is semi-activated. It can only execute operations that can update its own state, but cannot affect the state of other components. That is, it can change the value of its own state variables, can perform consumer operations, and can execute facet operations with only `in` arguments and receptacle operations with only `out` arguments. This state can be used to support the primary-backup replication scheme used for fault tolerance.
- **Inactive:** This is a more strict version of the passive state described above. In this state, components may not generate or respond to any events. Any incoming events from other components will not be handled; only the deployment infrastructure is allowed to alter the state of the component by changing its component parameters.
- **Active:** In this state the component is fully activated and is performing its operations when triggered.

State transitions are managed by the F6IAP deployment infrastructure. Figure 3 illustrates the lifecycle of a F6COM component and its interactions with this deployment infrastructure, called the F6 Deployment Manager (shown as DM) and the Component Fault Manager shown as (FM).

2) Component Interactions: Fractionated spacecraft are intended to provide a cluster computing environment where space mission applications can lease resources on the cluster to support their mission operations. Since these applications are likely to require heterogeneity in their interaction semantics and since they are all developed using components, it was important for F6COM to support different interaction semantics.

To support different interaction semantics, a F6 component in F6COM can have four different kinds of ports: consumer port, publisher port, facet port, and receptacle port. A publisher port is a point of data emission and distribution; a consumer port is a point of data reception. All data published or consumed are strongly typed. These interactions are specified in the OMG Data Distribution Services standard [15].

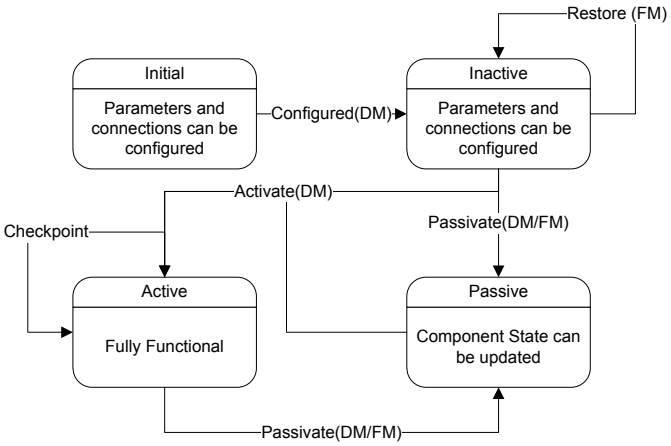

Fig. 3. F6 Component Lifecycle

A facet (of a server) is attached to the implementation of the methods defined in the provided interface and it services the requests issued through a receptacle on another component (a client) for these interface methods. Through these ports, three basic kinds of interactions can be realized: (a) anonymous, asynchronous, and non-blocking publish/subscribe, (b) synchronous call/return type point-to-point interactions, and (c) asynchronous method invocations by virtue of using facets and receptacles interacting asynchronously using call backs.

3) Timers and State Variables: F6COM also provides periodic and aperiodic time-based triggers that initiate component operations. Additionally, it supports state variables: component attributes with (limited) history, which are often needed in software interacting with physical phenomena. Their values represent a complete state of the component and they are often used by mathematical algorithms, e.g. Kalman filters for state tracking.

4) Extensible and Loosely Coupled Design using Connectors: The use of typed ports in F6COM dictates the interaction styles applications may use. However, ports alone do not decide the communication transport mechanism that will be used to implement these interaction styles. A number of choices are available. For example, synchronous call/return and asynchronous messaging can be supported using OMG's CORBA transport while the publish/subscribe mechanism can be supported using OMG DDS. A long term goal of the F6 program is to enable new transport mechanisms. Consequently, we had to decouple the transport mechanism from the structural artifacts of a F6 component, such as the ports.

Figure 4 illustrates the use of connectors [16], which decouple the transport and event handling mechanisms from the business logic of the ports. As shown in the figure, the component is clearly divided into two regions, the component executor region and the connector region. Typically, the component executor code is provided by the component developer and uses a local function call to interact with the connectors. The connector code is provided by the middleware vendor and together with the component business logic can be used to provide the different interaction semantics.

In our current implementation, the connectors are used

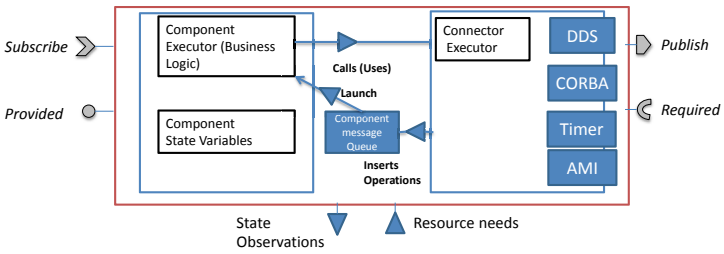

Fig. 4. F6 Connectors

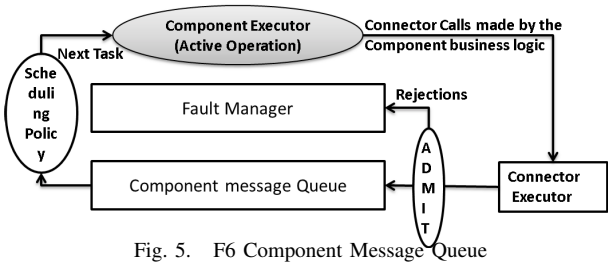

Fig. 5. F6 Component Message Queue

to implement the interface/request-reply style messaging (CORBA) as well as data-centric publish/subscribe style messaging (DDS). They are also used for client (using asynchronous message invocation, *i.e.*, AMI) and for servant (using asynchronous message handling, *i.e.*, AMH) are handled using an AMI connector. Timer-based events are handled through timer connectors. Components manage their state variables using the state variable connector.

The connectors are able to interact with the component executor region via the component message queue (CMQ), explained later in Section III-B5.

5) Component Operations — Promoting Race Condition-free and Deadlock-free Behavior: Robustness of mission-critical applications can be enhanced if the behavior is free of race conditions and deadlocks. Multiple threads with concurrent access to the internal state variables of the component will necessitate appropriate synchronization to be used. Such synchronization primitives often lead to unanalyzable code and can cause run-time deadlocks and race conditions.

The F6COM avoids such situations by breaking the different component activities into tasks or operations and ensuring that operations are scheduled one at a time and run to completion before another is scheduled. The component state can be updated only within the context of an operation. To implement the operation-based abstraction, the F6COM uses a dispatch queue that holds the ready operations, one of which is selected as next to run. The benefits of this decision are manifold: (a) application logic remains very simple, (b) there is no requirement for any synchronization primitives in the component code, and (c) the entire system is easier analyze for other properties of interest.

Figure 5 illustrates the Component Message Queue approach used by F6COM. Any incoming interaction request on a port (shown as middleware connectors in the figure), or an internal task generated due to timer expiration (shown as timer connectors in the figure) is placed into the message queue. All operations are quantified with two parameters, priority and deadline. The deadlines are expressed in absolute time and are judged from when the operation was inserted into the message queue. Scheduling of requests for execution is done based on a configurable scheduling discipline; currently, Earliest Deadline First (EDF), First In First Out (FIFO), and Priority FIFO are supported.

Queuing the activation record of an operation involves an admittance check, set by the component configuration. For example, the admittance check can ensure whether the deadline of the newly queued operation is beyond an estimated deadline given all the pending activations in the queue. A separate thread in the component framework: the fault manager thread is notified if a task is rejected. Once put into the queue, the tasks are sorted based on the queuing discipline. The component's main thread (component executor) picks the activation record from the queue based on the configured scheduling policy and runs the operation to completion. Operations must terminate in a finite amount of time and cannot be preempted by another operation of the same component.

6) Contract-based Programming for Robust Behavior: F6COM supports contract-based programming, which is necessary to realize robustness properties that are key objectives. To enable these capabilities, F6COM allows a component developer to specify pre-conditions and post-conditions that can be injected into the call path of the associated operation, such as publish and subscribe operations, as well as method invocations. These capabilities are as follows:

- **Pre-conditions:** This is a function with a Boolean return value, supplied by the component developer. It specifies a condition that must be satisfied before the operation is performed. This is typically used to build software anomaly detectors that can evaluate guard conditions over the current and historical values of input parameters of the operation, as well as current and historical values of the state variables of the component.
- **Post-conditions:** This is similar to a pre-condition, with the difference that a post-condition is checked after the operation completes.
- **Invariants:** F6COM supports the concept of state variables with history as it is common to store and process historical values in software components implementing mathematical algorithms that are likely to exist in the software interacting with physical phenomena. This history can be used to describe temporal properties over the component's state that must always be satisfied. Such conditions are expressed as invariants, and provide a way to detect anomalous conditions that violate the safety assumptions and are always evaluated when the state variables are updated.

Pre- and post-conditions, as well as invariants can be supplied by the developer as hand-crafted code, or auto-generated from models. In summary, together with invariants, pre-conditions and post-conditions specify the contracts that must be valid for components during runtime. Although these concepts are not new, their inclusion in the context of all other capabilities that F6COM supports makes this component model attractive. Note that the total time taken by a component operation will

be the time needed to complete the operation's business logic plus the time taken to evaluate the pre- and post-conditions.

7) First Class Support for Non-Functional Properties: Supporting nonfunctional properties, such as timeliness, fault tolerance and security are not an afterthought but rather an integral part of the F6COM design. For instance, every operation on a component can be associated with a deadline that the developer can specify. A developer can also specify time-based triggers that determine when selected component operations will be scheduled.

Section III-B4 described how the connector mechanism decouples the trigger type from the trigger port in a F6COM component. A special mechanism described below tracks whether the operation will meet its deadline or not. It also describes what happens when the deadline is not met. Timeliness assurances in F6COM are provided through two mechanisms: deadline *checks* and deadline *monitoring*. A deadline check is a mechanism that is performed before the operation is actually executed and is invoked at the following stages: (a) Before the activation record is queued, the system checks the current state of the queue, the queuing discipline used, and determine if this operation can finish before its deadline, and (b) Before the activation record is processed and the operation is actually executed, the system will determine if this operation can finish by its deadline. In either case, if the check fails, the fault manager (discussed next) is notified.

Deadline monitoring is invoked when the operation is allowed to execute and is accomplished as follows: the component container that releases the thread running the business logic of the component monitors the deadline. If a hard deadline is reached but the current operation is still active, then the framework notifies the fault manager. The fault management mechanism, in brief, is described below; detailed description, however, is not in the scope of this paper.

Fault management in F6COM is supported through a local fault manager, which is an integral part of the component. The fault management logic maps incoming anomaly events (precondition violations, post condition violations, deadline violations, admittance rejections) and maps them to preconfigured mitigation actions. In the past, we have shown how such fault management logic can be generated from timed state machine models [21], [22].

As part of the component implementation, the developers have to implement two interface operations `onCheckpoint (out OctetSequence)` and `onResume (in OctetSequence)` to serialize and de-serialize the state of the component, respectively, for use by the fault manager before checkpointing and after restarting. Checkpoint requests can be queued like any other operation. *Resume* can only be called if the component is in the Inactive state. In the Inactive state, the ready queue is empty. The "resume" operation will be pushed to this queue and immediately executed. Subsequently, the component can be brought out of the inactive state. These states are described in Section III-B1.

The underlying implementation to support the deadline monitoring and fault management involves three types of threads for a given F6COM component: a *Pusher* thread queues activation records into the ready queue as described in Section III-B5. It also monitors the currently executing operation for deadline violations, which can be caused by different factors such as high priority preemption or incomplete out-bound calls. The *Component Executor* thread runs the component implementation code, *i.e.*, picks the next operation to be activated (described in Section III-B5), and the *Fault manager* thread, whose operations were described above.

Security is handled using the concept of a security label (or a collection of labels) that are associated with the ports. Security labels determine the security classification(s) of the information propagated through the port, and they are assigned by some appropriate authority. These labels play a role in implementing support for Multi-Level Security (MLS). Although the security architecture in F6IAP is outside the scope of this paper, the core security mechanism works as follows. In F6IAP, the basic form of network communication (call/return or publish/subscribe) is through an operating system feature called Secure Transport. The basic communication unit in secure transport is *endpoints* (functionally similar to sockets) and *flows*.

A *flow* (configured by a suitably privileged actor) connects one source endpoint to one or more destination endpoints. When the business logic of an actor sends a message with a label through a port of one of its components, the underlying operating system (also developed in this project) checks the label of the message against the labels of the endpoint, which are securely stored inside the kernel and are known to be a subset of the labels of the actor. If the message label is not among the labels of the endpoint, the message is not sent (in the case of a writing-side endpoint) or delivered (in the case of a reading-side endpoint), but instead it is discarded. These checks are performed by the operating system and cannot be bypassed. Similar checks are also performed on the receiving end. See [18] for more details on secure transport.

8) Supporting Long Running and I/O-bound Operations: Space missions often may involve long running operations and multiple I/O bound operations (*e.g.*, sensor I/O). Since F6COM allows only one active operation to execute at a time within a component, it is possible that when an operation is waiting on I/O, a compute-ready operation may unnecessarily remain blocked. There are three options available to the component developer: (1) use blocking I/O, (2) use polling, and (3) use asynchronous I/O.

In the case of blocking I/O (every operation blocks until the transfer is complete) the component is unavailable while the operation is running. Other components may be running in the system, but the one waiting for the I/O is completely blocked. Through component-to-component interactions this blocking could propagate and introduce significant delays in the system. If polling is used, some activity has to be periodically scheduled that checks the completion of the I/O. Obviously this leads to a waste of resources and can lead to decreased performance. Hence, the only viable alternative is to use asynchronous I/O, where a component activity can

launch an I/O operation, not wait for its result, and when the result arrives, another activity — within the component — shall finish the operation.

Such asynchronous operations can be broken down into three activities as follows: (a) The starter activity that prepares an I/O operation and then informs the framework that the I/O operation can be scheduled, optionally passing data to the operation. (b) The I/O activity that actually launches the I/O operation and waits for its results. When the I/O completes the operation hands over the data (if any) resulting from the I/O operation to the framework and returns. This I/O activity is handled outside the context of the component and hence cannot impact the component state directly. (c) The handler activity that receives the result of the I/O operation and processes it, possibly forwarding it to another component.

Note that only the starter and handler activities are handled within the context of the component and interact with the component state and message queue. The I/O activity is handled by a connector that executes outside the context of the component and hence cannot impact the component state directly. When the operation returns, the I/O connector queues the request for activation of the handler activity.

This approach is needed to allow maximum concurrency in the component on one hand, and to ensure safety on the other hand. The activity starts and executes an I/O operation that can block. This blocking is acceptable because the component can still be used by other threads and all other components can be active. The physical I/O operation finishes and the rest of activity retrieves the data and hands it over to the framework for passing that to the handler. At this point the framework is free to release the handler activity that will be handed the data produced in the I/O activity. The handler will lock the component and can modify the component state, and can also propagate the data from the component to other components.

9) Resource-aware Allocation: Recall that F6 mission applications can span multiple compute nodes, spread across a potentially unreliable distributed network. These applications are realized as a workflow of F6 actors. Since the F6 cluster of fractionated spacecraft illustrates a highly resource-constrained environment, applications are not allowed to consume arbitrary amount of resources. Therefore, F6COM is supported by a platform and deployment infrastructure that follows fixed resource allocation scheme where the component's resource needs are declared at development time, verified at system integration time, and enforced at run-time.

IV. Illustrating F6COM Features on a Case Study

This section describes an example software assembly that illustrates some of the features of the F6COM component model. In order to show that the components always execute one operation at a time, we designed our experimental testbed to have three different components as shown in Figure 6. All three components run in their own actor and are co-located in a single physical node (for simplicity). The following is a brief description of functionality of each component:

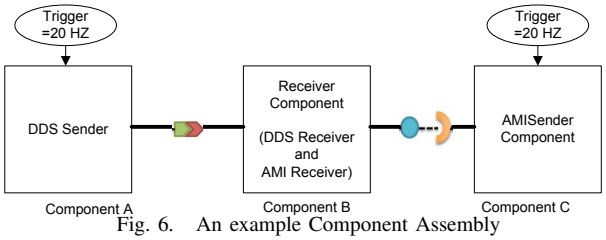

Fig. 6. An example Component Assembly

TABLE I
Source code evaluation (SLOC)

Component	Generated	Written	Total
AMI Sender	2121	289	2410
DDS Sender	2604	249	2853
Receiver	3230	242	3472

Component A: A DDS Sender that publishes a data instance every 50 milliseconds. **Component C**: An AMI (sender) client that sends Asynchronous Method Invocation requests every 50 milliseconds. The reply from the server is handled using the AMI callback operation. **Component B**: The receiver component with two ports: (a) DDS receiver port and (b) a server port that is used to handle the incoming AMI requests.

Table I shows the number of lines of code generated by the development tools such as IDL compiler and the number of lines of code written to implement the business logic of all three components. Approximately, 91% of the total code was generated.

Deployment and execution of all components results in a time sequence graph shown in Figure 7, which shows the activation periods of five different component operations — (a) DDS Send (DDS_S), (b) AMI Send (AMI_S), (c) AMI Sender Callback (AMI_SC), (d) DDS Receive (DDS_R), and (e) AMI Receive (AMI_R). It can be seen that AMI_R and DDS_R operations do not execute simultaneously. Both of these operation are executed in the same component (Receiver), however, we see that these operations do not overlap each other. This shows that the Receiver component always runs a single thread of operation. The AMI Sender component receives callback from the Receiver component when the AMI_R operation ends. The Receiver component receives DDS samples when the DDS_S operation starts but before the operation ends. This is because, the DDS Sender component performs some book keeping after publishing the data sample in the same operation.

V. Conclusions

Component-based Software Engineering (CBSE) is generally the preferred approach to develop large-scale, distributed systems. When CBSE is applied to develop distributed, real-time and embedded systems, the component model must support a number of features including robust application behavior that is free of race conditions and deadlocks while simplifying application development; first class support for multiple non-functional properties like timeliness, fault tolerance and security; dynamic resource management; full component lifecycle management. The F6COM component model presented in this paper supports all these capabilities. In addition, F6COM

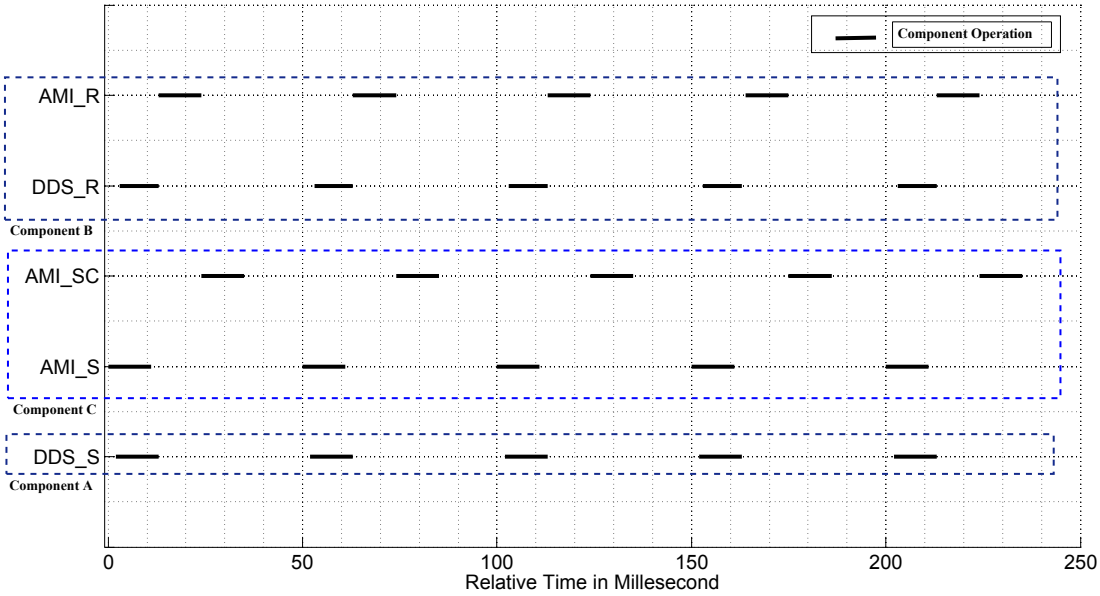

Fig. 7. A time sequence showing the activation of different operations in the assembly shown in 6. (DDS_S: DDS Sender, AMI_S: AMI Sender, AMI_SC:AMI Sender Callback, DDS_R: DDS Receiver, and AMI_R: AMI Receiver)

also supports effective resource sharing and isolation among applications, as well as allows applications to use different communication semantics. A qualitative evaluation of the capabilities of F6COM validates our claims about its design. Although F6COM has been designed for the fractionated spacecraft operating environment, it is suitable for many other kinds of distributed and embedded environments. In future, we intend to demonstrate its capabilities in a variety of cyber-physical environments.

Acknowledgments: This work was supported by the DARPA System F6 Program under contract NNA11AC08C. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of DARPA. The authors thank Olin Sibert of Oxford Systems and all the team members of our project for their invaluable input and contributions to this effort.

REFERENCES

[1] G. T. Heineman and W. T. Councill, Eds., *Component-based Software Engineering: Putting the Pieces Together*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[2] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich, "Components for Embedded Software: the PECOS Approach," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2002, pp. 19–26.

[3] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proc. of the International Symposium on Distributed Objects and Applications (DOA)*, vol. 3291. Agia Napa, Cyprus: Springer-Verlag, Oct. 2004, pp. 1520–1537.

[4] T. Bures, J. Carlson, S. Sentilles, and A. Vulgarakis, "A Component Model Family for Vehicular Embedded Systems," in *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*. IEEE, 2008, pp. 437–444.

[5] A. Dubey, G. Karsai, and N. Mahadevan, "A Component Model for Hard Real-time Systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517–1550, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1083

[6] O. Brown and P. Eremenko, "The value proposition for fractionated space architectures," in *Space 2006*, San Jose, CA, Sep 2006, pp. AIAA 2006–7506.

[7] "System F6." [Online]. Available: http://www.darpa.mil/Our_Work/tto/Programs/System_F6.aspx

[8] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[9] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava, "Comparison of Component Frameworks for Real-Time Embedded Systems," in *Component-Based Software Engineering*, ser. Lecture Notes in Computer Science, L. Grunske, R. Reussner, and F. Plasil, Eds. Springer Berlin / Heidelberg, 2010, vol. 6092, pp. 21–36.

[10] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. Van Den Born, "A Component Model for Field Devices," *Component Deployment*, pp. 1–13, 2002.

[11] J. Kim, O. Rogalla, S. Kramer, and A. Hamann, "Extracting, Specifying and Predicting Software System Properties in Component based Real-time Embedded Software Development," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 28–38.

[12] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2004, pp. 131–162.

[13] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.

[14] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[15] Object Management Group, *DDS for Lightweight CCM Version 1.0 Beta 2*, OMG Document ptc/2009-10-25 ed., Object Management Group, Oct. 2009.

[16] W. R. Otte, A. Gokhale, D. C. Schmidt, and J. Willemsen, "Infrastructure for Component-based DDS Application Development," in *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, ser. GPCE '11. New York, NY, USA: ACM, 2011, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/2047862.2047872

[17] *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, ARINC Incorporated, Annapolis, Maryland, USA, Jan. 1997.

[18] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, and W. O. et al., "A Software Platform for Fractionated Spacecraft," in *Proceedings of the IEEE Aerospace Conference, 2012*. Big Sky, MT, USA: IEEE, Mar. 2012, pp. 1–20.

[19] OMG, "Deployment and Configuration Final Adopted Specification." [Online]. Available: http://www.omg.org/members/cgi-bin/doc?ptc/03-07-08.pdf

[20] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer*, vol. 30, no. 4, pp. 110 –111, apr 1997.

[21] N. Mahadevan, A. Dubey, and G. Karsai, "Application of software health management techniques," in *SEAMS*, 2011, pp. 1–10.

[22] ——, "Architecting Health Management into Software Component Assemblies: Lessons Learned from the ARINC-653 Component Model," in *ISORC*, 2012, pp. 79–86.