A MODEL-DRIVEN SOFTWARE COMPONENT FRAMEWORK FOR FRACTIONATED SPACECRAFT

Abhishek Dubey[(1)], Aniruddha Gokhale[(1)], Gabor Karsai[(1)] , William R. Otte[(1)], Daniel Balasubramanian[(1)], Sandor Nyako[(1)], Johnny Willemsen[(2)]

[(1)]Institute for Software-Integrated Systems, Vanderbilt University, 1025 16th Avenue South, Nashville, TN 37212, USA, +1(615)343-7472, {dabhishe,gokhale,gabor,wotte,daniel,snyako}@isis.vanderbilt.edu

[(2)]Remedy IT, 2650 AC Berkel en Rodenrijs, The Netherlands, jwillemsen@remedy.nl

Abstract: *Fractionated spacecraft is a novel space architecture that uses a cluster of small spacecraft modules (with their own attitude control and propulsion systems) connected via wireless links to accomplish complex missions. Resources, such as sensors, persistent storage space, processing power, and downlink bandwidth can be shared among the members of the cluster thanks to the networking. Such spacecraft can serve as a cost effective, highly adaptable, and fault tolerant platform for running various distributed mission software applications that collect, process, and downlink data. Naturally, a key component in such a system is the software platform: the distributed operating system and software infrastructure that makes such applications possible. Existing operating systems are insufficient, and newer technologies like component frameworks do not address all the requirements of such flexible space architectures. The high degree of flexibility and the need for thorough planning and analysis of the resource management necessitates the use of advanced development techniques. This paper describes the core principles and design of a software component framework for fractionated spacecraft that is a special case of a distributed real-time embedded system. Additionally we describe how a model-driven development environment helps with the design and engineering of complex applications for this platform.*

Keywords: *Space computing, software components, model-driven software development*

1. Introduction

Fractionated spacecraft operated as a re-purposable multi-application platform for varying missions poses a number of challenges to software developers: in addition to providing specific functions (e.g. sensor data processing) the software (1) has to manage scarce computational and communication resources, (2) has to provide a framework for managing faults (both in the hardware and the software), (3) has to provide the required quality of service (QoS) to components implementing the services, and (4) has to provide the proper security isolation between users to ensure the confidentiality of information flows. Arguably, such challenges can be addressed by a platform-based architecture where a reusable and highly configurable, distributed software platform provides services to varying software applications that implement specific functions.

A software platform provides infrastructure to build applications, and many such platforms are in use today. The COM (and .NET) platform of Microsoft is the foundation for (almost all) applications running on Windows, while the industry-effort AUTOSAR is a similar example for Embedded Control Units (ECUs) for cars. Software platforms provide a framework in which the architecture of a (typically distributed) application can be explicated, and for this reason we are going to use the term 'Information Architecture Platform' (IAP) for the software platform.

The major ingredient of an IAP is middleware: a software layer that provides core abstractions and services to application code, and is running on top of some operating system (OS). To give an example, an OS may provide receive and send operations for messages, but the middleware may render a remote method invocation as a service to the application. A further useful abstraction is that of software components: reusable and executable units of software with well-defined interfaces and behaviors. Components interact only through their interfaces that are 'connected' on a higher level.

The key benefits to using middleware and software components are the abstraction of the end-to-end communication dependencies and interaction semantics, and the resultant execution of component behaviors. Additionally, the use of software components and middleware enables location transparency, which makes it easier for components to be migrated across different nodes at runtime. Overall, these well-defined patterns along with other supported features, such as resource management and support for quality of service attributes, allow an application to be viewed as a composition of reusable components. When each component is developed separately and is connected to other components using the provided patterns, it yields well-architected, useful, portable, and easily-managed applications.

In this paper, we focus on the software component framework and describe its key features, architecture and design. We call the software component framework FX for 'fractionated spacecraft'. First we introduce an important background concept: the Interface Definition Language. Next we provide a brief overview of the full software platform, FXIAP. This is followed by a description of the component model and the development environment.

2. Background: The Interface Definition Language (IDL)

A central tenet of systems and software engineering is the importance of interfaces. For systems and software interfaces are essential and their precise definition is critical. In software interfaces need to be defined in a manner independent of the implementation language used, and this is accomplished by the industry standard Interface Definition Language (IDL). IDL provides for defining data types, constants, exceptions, and operational interfaces: collections of named methods, with input/output arguments.

IDL interfaces define a contract between a client and a server. A server is said to *implement* an interface, as it supplies an implementation for each of its methods via an object. Note that interface implementations are typically stateful. If a client obtains a

reference to the server's implementation of the interface (i.e. the object), then, in the knowledge of syntactical specification of the interface, it can invoke the methods on the object. One server object can implement multiple, distinct interfaces. A server is said to expose *facets*: handles to interfaces that the server implements. A client is said to have *receptacles*: handles to interfaces a client requires. Receptacles and facets are connected (i.e. a particular client is linked to a particular server) when the system is configured. Note that the client can be implemented independently from the server (as it only has to know the interface that is accessible via a receptacle), as the binding between the client and the server happens late in the process, at run-time. This is approach underlies all component-based software construction: clients and servers a typically roles that components play, and the applications are constructed at run-time by 'wiring' the components. Note that the same mechanisms provides also for fault tolerance.

IDL can be used to define data types, specifically structs (i.e. structures). Such structs can be used as message types in message-oriented distributed systems. In order to establish identity of messages, fields in the structs can be marked as 'key' indicating that a new value in that field means a distinct, new instance of the data item. The same key value means an update to an already existing data item. Note that such message-based distribution means a loose coupling between data providers (publishers) and data consumers (subscribers): publishers are unaware of their subscribers, and some infrastructure (i.e. the data distribution middleware) is responsible for dispersing the data.

In either of the above cases, the IDL language has implementation-language specific bindings. Such bindings are implemented by the IDL compiler that translates IDL into the implementation language (e.g. C++ or Java) such that it could be integrated with the middleware libraries (to be discussed below) and code supplied by developers.

3. The FX Information Architecture Platform

The Information Architecture Platform (IAP) is the conceptual and technical underpinning of all software that runs on the distributed hardware platform. The IAP provides essential services and functions for end-users. As such, it is responsible for managing resources, providing a distributed computational platform for running mission-specific applications, implementing information security services, managing faults in the system, and provisioning all design-time tools to build, deploy, and operate the applications.

The IAP is a layered architecture. The *operating system* provides core services required for node operations. These include process management, concurrency, synchronization, and device management services. The *middleware* provides higher-level services supporting request/ response and publish/subscribe interactions for distributed software. The *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. The middleware is layered

upon the operating system, and the component model is implemented on top of the middleware.

Additionally, IAP also provides a set of system-level *platform services* that are required to coordinate the software lifecycle, mission operation and fault tolerance across the set of spacecraft in the cluster. These services are:

1. Deployment Management: The deployment manager is responsible for installing component applications on each node and managing the lifecycle of the applications. It is similar to the command line shell of conventional operating systems, but it does not have a 'user interface'.
2. Operations Management: The operations manager is responsible for the overall management of operations on a node or the cluster. The operations manager receives and interprets operational commands from the ground including starting and stopping services and deploying new applications. There is one operations manager on each node in the cluster.
3. Dictionary Management: The dictionary manager is responsible for providing a repository for storing and looking up application configuration data across the cluster. It acts as a global directory for all active components in the cluster.
4. Fault Management: The fault manager is responsible for providing fault detection, isolation, and recovery services to applications.

A detailed discussion of these platform services is out of scope for this paper; more details can be found in [1].

4. The FX Software Component Framework

The IAP provides support for a reusable unit of software (a software component) with well-defined interfaces, execution semantics, and interaction semantics for application developers. This requirement is to promote abstraction, modularity, and reusability, to increase the productivity of developers, and to ensure that requirements of the applications and of the system can be mapped to the capabilities of the components.

Applications are constructed by instantiating components and composing them by configuring their interconnections. Interconnections mean 'interactions' that permit synchronization and communication between activities of software components and the platform.

The nominal execution semantics of an FX component depends upon (a) the interaction patterns specified on the component, and (b) the timing properties specified on the component.

4.1. Interaction Patterns

Systems composed of components must provide for interactions among components to implement system functions.

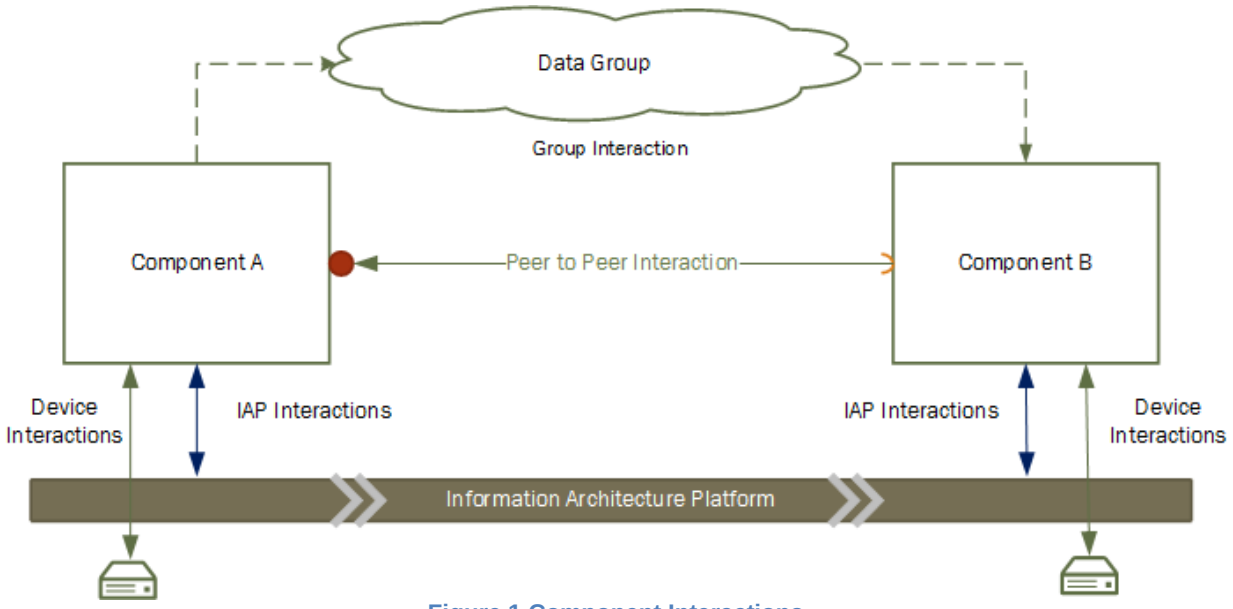

Figure 1 Component Interactions

Figure 1 illustrates the different interaction patterns that a component can participate in. These interaction patterns can be generally broken down into three categories (a) Interactions with other components, (b) Interactions with IAP, and (c) Interactions with Payload Devices.

4.1.1. Inter-Component Interactions

The component model prescribes two different classes of communication interaction between components (a) Peer to peer and (b) Group communication. While peer to peer communication is useful whenever tight coupling and synchronization if required between two components, irrespective of their location, the group communication pattern is intended to decouple components from each other. Thus, communication can be established over an emergent “global data space” – to which components can either write to or can read from. These two patterns are described by two different Object Management Group Standards, CORBA [2] and DDS [3].

4.1.1.1 Peer to Peer Interactions

When tight coupling and strict synchronization between components is required, an interaction pattern with call-return semantics is desirable. In such situations the caller who made the request is blocked until the call successfully returns, an error is detected, or a timeout occurs. This is the most common interaction pattern as it is equivalent to method invocation and is often referred as Remote Method Invocation.

```
interface DeltaVSetting
{
    Status setDeltaV(in DeltaV deltaV, in Time when);
};
```

For example the *DeltaVSetting* Interface shown above has a method for setting Delta Velocity. This interface will typically be provided by a Thruster Control Component, while an orbit controller component will invoke this interface.

An alternative to synchronized execution of remote method invocation, where the client is blocked till the response arrives is called asynchronous method invocation (AMI). In this method, the client component is not blocked waiting for a response from the server, rather the response is handled via a callback at a later time. This allows better utilization of resources and more responsive behavior.

4.1.1.2 Group Interactions

The group interaction between components is realized via the publish/subscribe (pub/sub) paradigm. This is a group interaction pattern, wherein multiple publishers and subscribers communicate within a specified `domain' over a specified `topic name'. Topics are always typed and associated with a data structure. Different values associated with the same topic are called samples. The data model allows for definition of topic instances, which partitions the data samples into equivalent classes based on the value of key fields (similar to primary keys in databases). A topic type without any primary keys is said to be event. Events are like singleton classes where there is only one instance of the class, but the value of the data members can change. Topic keys imply that there can be different object instances of the topic with a different value assigned to each instance.

Unlike peer to peer interactions, pub/sub interactions can be used to establish data exchange between number of components - multiple writers and multiple readers. Furthermore, the group of interacting components can change over time, i.e. publishers and subscribers can join or leave the group.

```
struct State {
  double x,y,z;
  double x_dot, y_dot, x_dot;
  Time time;
};

struct SatState {
   unsigned long satId; //@key
   State state;
};
```

For example, consider a group of navigation component, one on each satellite. These components will publish the state of their satellite "`SatState`" and consume the state information from other satellite without actively seeking out any one particular satellite for communication.

4.1.2 Interaction with IAP

Over the lifetime of the system it is expected that applications will be dynamically deployed, started, stopped, restarted, and removed from the system. A deployed application may be started, stopped, and restarted many times. The orderly execution of such management operations necessitates support from the component framework.

All components are hosted within a container that is provided by the IAP and manages the lifecycle of the component. At any time during its life a component can be in one of the following states[1]:

- 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 remote method invocations where it cannot affect the state of other component it is interacting with. This state can be used to support the primary-backup replication scheme used for fault tolerance.
- Inactive: This is a stricter 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.

The other interaction between IAP and components is related to resources. The components are allowed to request access to storage resources such as disk storage, and memory up to the specified maximum quota, or access to computational resources such as CPU cycles. Note that component code does not directly access either the threading or synchronization primitives, these are managed by the other IAP layers; notably the middleware. We will discuss the threading and concurrency in a later section on execution semantics.

[1] These state transitions are managed by the IAP services.

In addition to resources and lifecycle management, a component can also register periodic and aperiodic time-based triggers with the platform. These triggers are in fact timers that invoke a component operation upon timer expiration.

Lastly, components interact with IAP for state management. In order to facilitate anomaly detection and fault management on the components, their state must be observable by an external entity responsible for fault management. Additionally, components supports state variables: component attributes with (limited) history, which are often needed in software interacting with physical phenomena.

4.1.3 Interactions with Payload Devices

The FX component framework enables the components to access payload interfaces. Each payload device (managed via a device driver in the operating system) is associated with a dedicated software component at runtime. This association is managed and controlled by the IAP. Once a payload is associated with a component, the component can send data to the devices. By default, the send operation is non-blocking and facilitated by the component framework. The result of the send operation or an up call from the IAP due to a device driver activity is sent to the component business logic using a callback provided by the component developer.

4.2 Component Execution Semantics

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 necessitate appropriate synchronization to be used. Such synchronization primitives often lead to unanalyzable code and can cause run-time deadlocks and race conditions.

The FX component framework avoids such situations by breaking 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 altered only within the context of a component operation. Typically, the business logic code for a component operation is provided by the component developer.

The component operations can directly invoke the operations provided by the framework library. The framework operations typically do not block and return immediately, except in two cases: (a) synchronous remote method invocation and (b) a get or a waiting read performed by a subscriber. In both cases, the framework operation call blocks until either a response is available or a timeout occurs. To specify the timeout, all framework operations are marked with a timeout parameter, which can be configured when component is in the initial state.

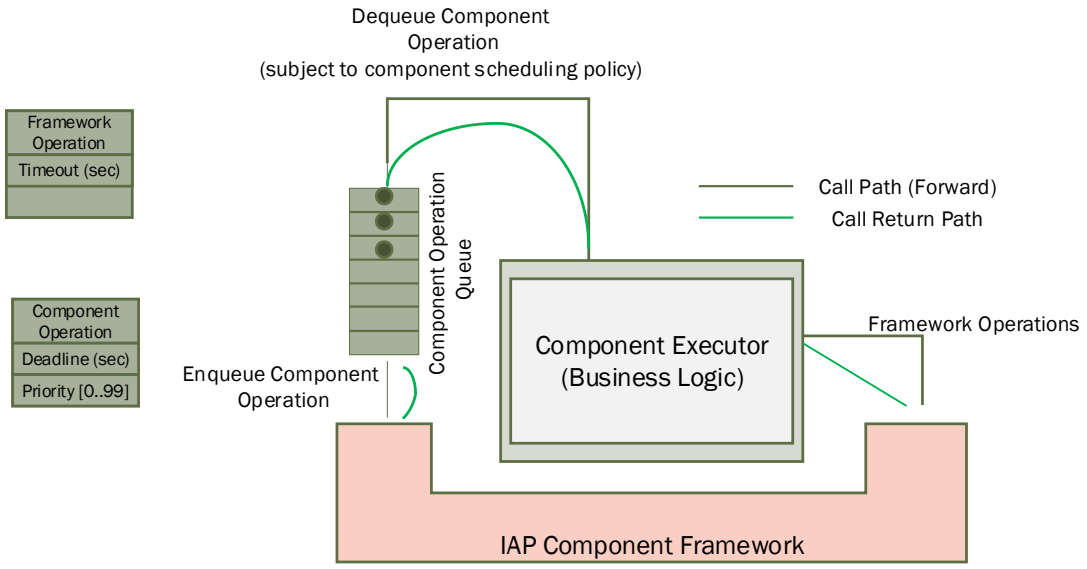

Figure 2 Component Execution Sequence

Figure 2 shows the interactions between the framework and the component executor: i.e., the business logic. Component operations invoked by the framework are queued in the component's operation queue. The framework issued enqueue operation returns immediately to the framework. The scheduler for the component operation queue selects the next component operation to be executed based on a configurable scheduling policy. Currently, we support two policies: Priority First in First Out, and Earliest Deadline First. Both of these policies are non-preemptive, i.e. once an operation started it cannot be interrupted. However, it is possible to monitor deadline violations for the operation: if an operation takes longer than expected then a fault management action could be triggered to start a mitigation activity.

Additionally, a callback function associated with a timer can be enqueued as a component operation. The code for this operation is provided by the component developer and it becomes eligible to run when the timer expires. This supports the time-triggered scheduling of component operations. In (soft) real-time systems operations can be event-triggered or time-triggered, the latter meaning that an operation is activated because of the elapse of time. Such scheduling of operations is essential for implementing periodically triggered calculations for control applications, to activate scheduled computations at specific points in time, and to implement timeouts for operations to detect and manage faults.

To support these scheduling policies two attributes for each component operation must be configured: deadline (i.e. anticipated worst-case execution time), and an unsigned integer priority. For normal operations priorities must fall within an closed interval (e.g. [0..99]), and while priorities greater than the larger value of the interval are possible they are reserved for component operations used to implement system-level operations for the lifecycle and fault management services interactions.

4.3 Distributed Applications

While a single component provides a reusable software unit that implements one or more functions, an application can potentially consist of multiple components. To provide memory address separation between different components, components are grouped together into *actors*.

Actors are similar to processes in traditional operating systems. The key difference is that actor identities persist across process restarts and that actors are uniquely identifiable across all the computing nodes in the distributed cluster. A group containing one or more actors deployed together that work collaboratively forms an *application*. An application may be split across many actors potentially distributed across on different nodes.

5. Model Driven Application Design

Although the IAP provides run-time support for actors and their components, developing, deploying and configuring the actors and their components is fraught with both accidental and inherent complexities. The accidental complexities arise from the tedious and error-prone nature of the process used to (a) compose components to form actors, (b) deploy the actors in the nodes of spacecraft modules, and (c) configure the actors for their 'quality of services' (QoS) properties. The inherent challenges stem from multiple sources. First, command-line based deployment and configuration of applications is not feasible in remote deployments (i.e. in space). Second, intermittent and highly fluctuating network connectivity implies that ground-controlled orchestration of applications on the fractionated spacecraft is not an option. Third, scheduling and admission control of actors is based on runtime availability of resources and security policies. Owing to such factors, significant autonomy is desired in handling the lifecycle of actors and their components. Manual approach is neither feasible nor would it provide the stringent assurances on correctness properties for the mission critical applications.

Designing and constructing distributed applications for such a complex and powerful software platform is therefore extremely challenging: there are many accidental and inherent complexities that developers have to cope with. We address these challenges by using a model-driven software development environment (MDE) that application developers can use to build components and then compose them to architect and configure applications. For an overview of the MDE and the process see Figure 3.

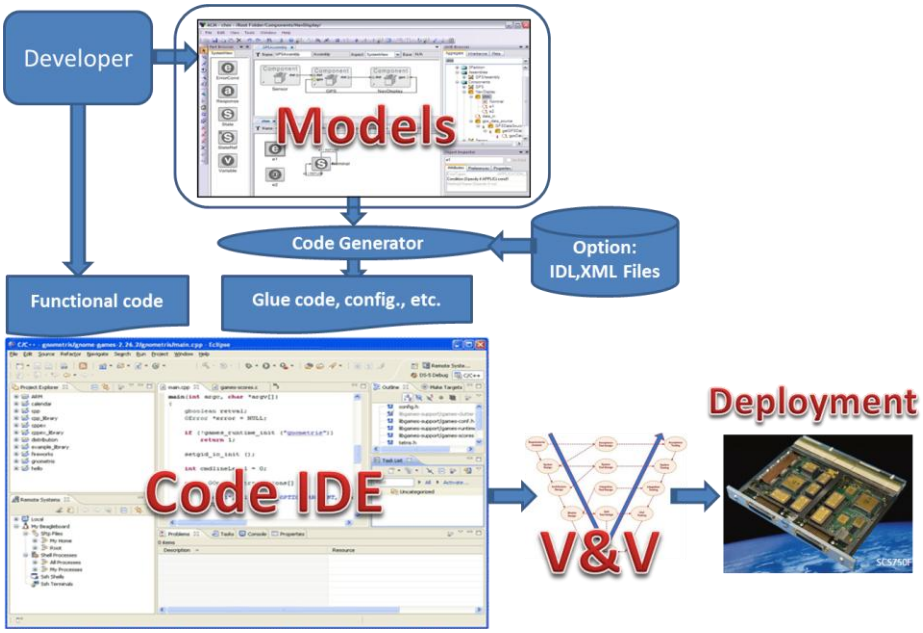

Figure 3 Model-driven Development Environment

The MDE is based on a domain-specific modeling language that allows (1) modeling of components, actors, and applications together with their interfaces and interactions, (2) how the applications are to be deployed on the platform, (3) security labels and their association to component information flows, (4) QoS properties and resource needs of the application and its components, and (5) hardware platform architecture and configuration. Note that component implementation is not modeled – this can be implemented in C++ or in another model-based tool (like Simulink/Real-time Workshop) [4] that is capable of producing executable code. Configuration files and glue code are generated from the domain-specific models, and the code is compiled and linked to implementation code to form binaries for components, which are then deployed on the platform using the mechanisms provided. The models also permit design-time analysis: system integrators can make admittance decisions based on the models. Such analysis is needed to decide if the resource needs of the applications can be satisfied on the space-based platform.

5.1 Example

5.1.1 Application Development

Let us consider an example satellite flight control application. Assume that this application controls the thruster on each satellite such that the satellite follows a preplanned orbit, while maintaining a formation. In order to build this application, an application developer can choose to use a number of preexisting components[2].

[2] We do not show the component development stage in this example. However, the FX Modeling language supports component development and maturation as a separate activity. Once a component is developed, an application developer can choose to use existing component or build a new one.

Figure 4 and Figure 5 show the model of an example satellite flight application. This application is installed on each satellite in the cluster. The application consists of six different components:

1. GPS – This component manages the GPS payload and publishes periodic GPS information update. This component also provides position update via remote method invocation (i.e. a query).
2. IMU – This component manages the Inertial Measurement Unit and provides the inertial state of the satellite.
3. NavFilter – This component uses the GPS and IMU information to provide filtered inertial state of the satellite, including its position.
4. Orbit Planner – Provides the planning service that computes the planned trajectory.
5. Orbit Controller – This component consumes the current position, the planned trajectory and computes the necessary 'delta V Required' commands.
6. Thruster – This component manages the Thruster device. It consumes the delta V commands and actuates the thruster.

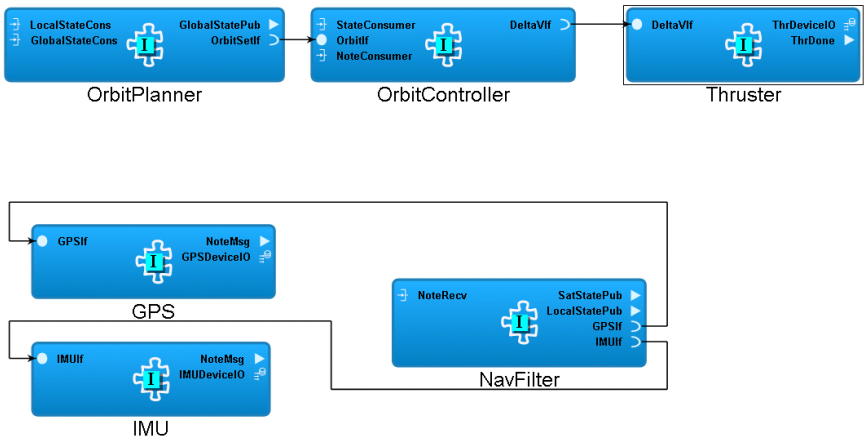

Figure 4 Peer to Peer Interfaces in the Satellite Flight Application

Note that the arrows in Figure 4 show the direction of peer to peer call-request interaction. Publish-subscribe interactions are shown as ports (▶ publisher, ⊣ subscriber) without any arrows associated to them. Figure 5 shows how the application components are mapped to actors. Recall that actors provide a separate address space, similar to processes in traditional operating systems.

5.1.2 System Integration

Consider a satellite cluster with three different satellites: Alpha, Bravo and Charlie. The system integrator can specify the network configuration of these satellites and how the network configuration might change over time in the modeling language. Additionally, system integrator can model the deployment of software on each satellite.

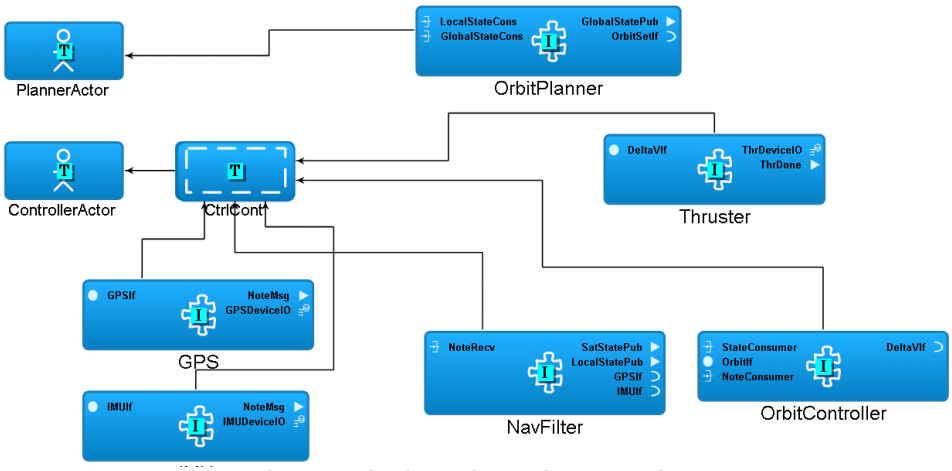

Figure 5 Single Node Deployment View

Figure 6 shows the deployment and operating system schedule definitions for three satellites. Note that the underlying operating system supports temporal partitioning and the 'schedule definition' provides the release times for the partitions within a hyperperiod. The modeling tools enable a system integrator to reuse the flight application (`SWPackR`) and deploy it on all the three nodes.

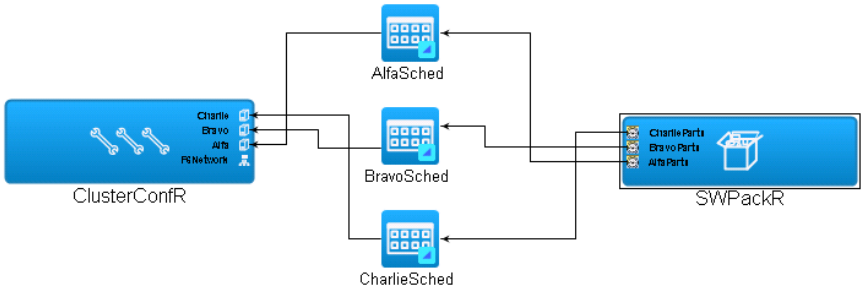

Figure 6 Deployment of Flight Application on Three Satellites

6. Related work

In [7], 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) [8] 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 FxCOM shares the same goals as PECOS: it is tailored to support multiple nonfunctional properties in resource-constrained and highly uncertain environments. FxCOM 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 FxCOM and PECOS. FxCOM 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, FxCOM also supports time-triggered actions.

The GENESYS (GENeric Embedded SYStem) [6] research project has developed a cross-domain component-based architecture for embedded systems. It has been designed for achieving (1) compositionality to allow system designers to compose systems using independently developed and tested components (unit of abstraction), (2) robustness to provide fault containment and selective restart of components upon failure, and (3) energy efficiency by integrating resource management in the platform design. An important principle followed in GENESYS architecture is the strict separation of computational components and communication paradigm

The AUTOSAR standard [5] specifies how application developers and system integrators from different companies can work separately and independently from underlying implementation of in-vehicle communication technologies and yet achieve compositionality. Moreover, it allows designs that can be rapidly reused and deployed on different kinds of hardware.

In comparison, FxCOM provides a richer set of communication interactions and component states. It also supports dynamic deployment and configuration of components at runtime.

7. Conclusions

In this paper we introduced a software component framework, and it's supporting run-time and design-time elements. A prototype of the component framework is being developed and tested on various applications. We provided an informal definition of the component model behind the framework that is based on precisely defined interaction patterns. Applications are constructed by composing components and concretizing (i.e. configuring) the component interactions. This latter process is supported by a model-driven development environment.

The model and framework described above provides a powerful development environment for constructing distributed, real-time, embedded applications that run a set of networked nodes: i.e. the typical architecture of a fractionated spacecraft. The strengths of the component framework come from the use of interactions as the key abstraction for composition, while the model-driven environment offers strong abstractions to manage complexity: explicit component models with interfaces, models for configuring the deployment details of the system, etc.

An earlier prototype of a similar component framework has been used to develop the main flight software algorithms for a fractionated spacecraft. That experiment has shown the power of component-oriented development and the best ideas were carried forward in the work described. The actual evaluation of the component framework on larger examples is ongoing work, but early results clearly indicate acceptance by developers.

8. Acknowledgements

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.

9. References

[1] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose, “A Software Platform for Fractionated Spacecraft,” in Proceedings of the IEEE Aerospace Conference, 2012. Big Sky, MT, USA: IEEE, Mar. 2012, pp. 1–20.

[2] http://www.corba.org/

[3] http://www.omg.org/spec/DDS/

[4] http://www.mathworks.com/products/simulink/

[5] AUTOSAR Technical Overview. [Online]. Available: http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf

[6] http://www.genesys-platform.eu/

[7] Petr Hošek, Tomáš Pop, Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. 2010. Comparison of component frameworks for real-time embedded systems. In Proceedings of the 13th international conference on Component-Based Software Engineering (CBSE'10), Lars Grunske, Ralf Reussner, and Frantisek Plasil (Eds.). Springer-Verlag, Berlin, Heidelberg, 21-36

[8] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. 2002. A Component Model for Field Devices. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment* (CD '02), Judy M. Bishop (Ed.). Springer-Verlag, London, UK, UK, 200-209.