Container Services for High Confidence Software

Gary J. Vecellio, William M. Thomas, and Robert M. Sanders

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508

{vecellio,bthomas,rsanders}@mitre.org

Abstract. For high-confidence systems, substantial effort is expended to ensure that the system will behave in an expected manner. This paper presents the position that a commercial off the shelf Enterprise JavaBeans™ (EJB™) container can be extended to constrain the behavior of high confidence software. These extensions are based on techniques that are not new; developers of high confidence software use these techniques already. We discuss how adding services to the container allows verification and validation of high confidence properties to be performed at the container level where they will require less revalidation when business logic changes. We present two examples of container augmentations, one based on communication mediators and one based on application monitors. These examples illustrate how method-level preconditions and postconditions, and application-level invariants can be asserted by an EJB container.

1. Introduction

For high-confidence systems, assurance is needed that the system will behave in an expected manner. Since software is being used in ever-increasing levels in systems where the consequence of failure is high, the role software plays in such systems must be addressed. This creates a need for dependable, reliable, secure and survivable software, i.e., "High Confidence Software" (HCS). There are a variety of techniques that have been employed to help deliver HCS, including process-related standards such as RTCA DO-178-B or MIL-STD-882-C, as well as design patterns and approaches providing capabilities such as application monitoring, constraint checking and policy enforcement. Such techniques have been successfully applied to a variety of projects, with the drawback being that they are generally expensive to employ, and are not tailored to efficiently support modern software development approaches (e.g., component-based software development).

Component-based software (CBS), in particular as realized in a standardized framework, such as Java™ 2 Platform Enterprise Edition (J2EE™), has seen significant success in enabling rapid, cost-effective development and deployment of e-commerce software. This is likely in part a result of the services offered by J2EE, and in particular, the services offered by the Enterprise JavaBeans™ (EJB™) container. The EJB container offers services such as transactions, security, connection pooling, object passivation, among others. These services often can be leveraged by the application developer and integrator to reduce effort. There is increasing interest in techniques that offer even more flexibility, such as techniques to facilitate independent extensibility, support for hot-swapable components, and increasing levels of

declarative programming. Such techniques have the potential to demonstrate even greater benefit to the application developer, integrator, and deployer, but increase the burden to ensure that the application remains in an expected state.

We believe that services can be added to a container that will provide the developer the mechanisms to efficiently compose a robust system with well-constrained behavior. A container seems to be an appropriate place to offer such services, as it provides a well-defined separation of the application logic from the framework service infrastructure. Our focus is less on the development of new mechanisms to support a general separation of concerns, but rather on how existing techniques can be tailored both for integration within a standard component framework and to support the specific needs of HCS. We have modified an open source EJB container to support such services. Our modifications fall into two categories: *mediators*, which control communications among components and between a client and server; and monitors, which examine the state of the application outside the context of component communication. Together, these mechanisms provide support for ensuring that an application starts in an acceptable state, maintains a known state, and in the event that it does transition to an erroneous state it can be transitioned back to an acceptable state. These modifications support the application of HCS techniques in the container and thus make the software easier to develop, modify, and maintain.

This paper provides an overview of some of the needs of HCS that are of interest in a CBS setting, describes how services supporting these needs can be implemented in the container, and presents high-level examples of how these services can support HCS applications.

2. High Confidence Software

One type of "high confidence" software is the software in safety-critical systems. Hermann provides a definition of Software Safety as:
"*… features and procedures which ensure that a product performs predictably under normal and abnormal conditions, [and] the likelihood of an unplanned event occurring is minimized and its consequences controlled and contained; thereby preventing accidental injury or death, whether intentional or unintentional* [2]."
This notion of predictable behavior seems fundamental to HCS, and a variety of design techniques have been developed and applied to such system that help to constrain behavior to a predictable range. However, as the complexity of such systems grows, the traditional techniques become less effective, in many ways due to their lack of support for modern complexity-reduction techniques, such as component-based software engineering. We need to adapt the traditional techniques to better support current development needs.

The Interagency Working Group on Information Technology Research and Development, in their report on "High Confidence Software and Systems (HCSS) Research Needs", notes the need for:

" … theoretical frameworks, engineering methods, and supporting tools that directly enable composition and decomposition (both integration and top-down design methods) with reasoning about consequences for crucial system properties. Sound composition and decomposition methods and sound engineering principles must be developed as a foundation for languages and tools that can be used to engineer robust systems. … Challenges include the following:

- *New approaches to component technology and sound composition of "peer" components coupled with design methods that weigh multiple aspects of a system and compose property-based measures to achieve overall system assurance.*
- *Support for managing hierarchically designed or layered systems. This includes reasoning and transformation that crosses layer boundaries: hardware, networks, operating systems, and applications. It also includes management across the boundaries of interacting, hierarchically structured subsystems.*
- *Support for explicit description and analysis of interactions between the system and its operational context. This would further enable reasoning about, and controlling, the interactions of software and systems with the environment or with human operators.*
- *Representation and management of "meta-descriptions" for recording and managing composite system structure."* [3]

This report highlights the need for infrastructure supporting robust, predictable, composition techniques and tools. We are investigating one approach to support such needs, namely, determining how existing container services may be extended so that they can better support the HCS needs of bounded, predictable behavior and identification of and recovery from "abnormal" behavior.

3. Container Support for HCS

Our approach to providing support for high-confidence software is based on adapting traditional HCS techniques so that they can be effectively used in component-based systems. Our integration mechanism for these HCS services is the container, and specifically, in our initial prototype, an EJB container. While our initial investigation has exclusively focused on EJB containers, we believe the concepts should apply equally to other types of containers.

Our focus to date has been on augmenting commercially available, open source, EJB containers and servers. Augmenting established technology should offer the benefit of ease of adoption, and allow for reuse of much of the existing container design and implementation. The focus for our proof of concept prototype has been to develop extensions to the JBoss[1] open source EJB container and server. Our extensions include definition of new container types, modifications to deployment

[1] See www.jboss.org or http://sourceforge.net/projects/jboss/

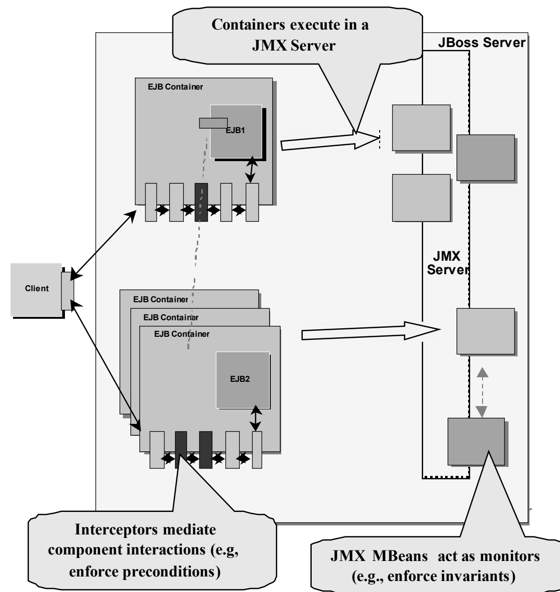

Figure 1. Overview of the JBoss Architecture

descriptor and server configuration files, and the development of mediators and monitors.

Figure 1 provides a high level view of the JBoss Server Architecture. The JBoss server is based on the Java™ Management Extensions (JMX) framework. JBoss EJB containers are built up dynamically and loaded into the JMX framework. The EJB containers themselves are based on plug-in framework, the JBoss Container Framework. This means there are essentially an unlimited number of different container types that can be defined and loaded into the server.

In the JBoss Container Framework, plug-ins implement specific interfaces and are selected by specifying them in an XML-based deployment descriptor file. Included in the plug-in set is the Interceptor interface. Interceptors are instantiated and chained together into a linked-list by the container factory. The last interceptor in the chain is the container itself, which invokes the business method of the EJB.

We augment a standard JBoss EJB container by plugging our interceptor-based mediators into the JBoss Container Framework. These augmentations can be accomplished at the level of the server (i.e., defining a new container type that all EJB can use) or at the EJB level (i.e., defining a new container type that is local to a specific EJB). We augment the EJB server by plugging MBean-based monitors into the JMX framework. This is a server level augmentation that allows the monitors to access the other JMX MBeans and all the containers and EJBs loaded into the server.

3.1 Mediator-based Services

Mediators are a mechanism to control communication between components and between clients and components. As such, they supply the capability to monitor data being delivered to and returned from components. This offers effective support monitoring component preconditions, postconditions, and invariants in a manner outside the context of the component. This enables the extension of component "contracts" to support "composition contracts", where the goal is to ensure that the composition of a set of components is realized in accordance with the composer's expectations, and that such composition does not violate the expectations of the individual components. For example, contract preconditions are evaluated before an EJB method is called and contract invariants and postconditions are evaluated before the results are returned to the client. This approach is similar to Design by Contract™ [6], but the assertions are evaluated by the container rather than the client and service. We see this as an advantage because it means the client is required to have less knowledge about the service, and it provides the flexibility to allow different contracts involving a single component (i.e., it extends the notion of contracts to "composition contracts" to more effectively support CBS.)

As an example, consider the software interlocks of a weapons system. Suppose we have an application that is composted from the following components: a Weapon Control (WC) EJB that has a Fire method that releases the weapon, an Identify Friend or Foe (IFF) EJB that determines the target nationality, a Threat Evaluation (TE) EJB that determines if the target represents a threat to allied forces, and finally, a Launcher (L) EJB that determines launcher status. The safety interlock for this application is to prevent the firing of the weapon in all cases except when:

- all IFF identify that the target is Hostile (as opposed to Friendly, or Unknown),
- all TE identify that the target is Closing (as opposed to Not Closing, or Unknown), and
- L is Ready (as opposed to Not Ready, or Error).

The safety interlock is implemented as interceptors in the WC EJB container decoupling the interlocks from the core application logic implemented in the IFF, TE, L, and WC EJBs. The WC container is parameterized at startup by the WC deployment descriptor, which specifies a sequence of interceptors that are activated when a call is made to the EJB. In addition, the deployment descriptor can specify the dependent EJB attribute that is checked and the expected result of evaluating the assertion.

The interceptors associated with the WC EJB receive control when the EJB is loaded, before the application begins execution, each time the WC EJB is called, each time the WC EJB returns a result, and finally when the application is shut down. When the WC interceptors receive control before the application begins execution they check the application environment for containers that contain the dependent EJBs

(i.e., IFF, TE, and L). References to these containers are stored in the interceptors for later use.

On a call to the WC EJB, the interceptor chain is activated. The first interlock interceptor will determine if the call is to the Fire method. If not, control is passed to the next interceptor. If it is the fire method, the interceptor will then determine from the IFF EJB(s) if the target is hostile. If all agree that the target is hostile, control will pass to the next interceptor in the chain. Otherwise, an exception is returned to the client. In a similar manner, the second interceptor verifies that the target is closing, and the third interceptor verifies that the launcher is ready. If all interlocks succeed (no interceptor returns an exception), control will pass to the next recipient in the chain (in this example, the next recipient is not one of the interlocks).

The interlock interceptor chain is transparent to callers of the Fire method and is decoupled from and transparent to the WC EJB. Modification to the interlocks can be accomplished by changing to the WC deployment descriptor or the interceptors. This can be done without impacting the client code, the WC EJB code, or dependent EJB. Deployment of a WC EJB with different logic in the same WC container will enforce the interlocks on the new WC EJB.

3.2 Monitor-based Services

Monitors provide a mechanism for observing application state outside the context of component invocation requests. Such monitoring is commonly used when designing for safety, e.g., the design of the "safety executive" described in [5] includes the use of watchdog processes that provide application-monitoring capabilities. One benefit of the safety executive approach is that it offers improved visibility and consistency of the monitoring and recovery features of the design. For example, monitors can be used to evaluate application level invariants at specified time intervals. This allows for the detection and correction of application-wide state abnormalities.

As an example, suppose we have an application that contains components that use services outside the control of the application (e.g., a dependency on a database that is located at an external location.) To function properly this application must be able to connect to and exchange information with the remote database, but this dependency on a remote database may not be known until the application is composed. That is, the database access interface can hide the fact that it is using a remote database to provide its service. In an assembly where there is such a dependency, other components in the application might not know that they have to account for the database component returning an error when it cannot access the remote database due to a network failure.

To allow the application to detect and recover from such an error, an "application-wide invariant" can be specified to monitor that the remote database remains accessible. This invariant can be enforced in the JBoss server as follows. A JMX MBean can be used to periodically check the status attribute of an arbitrary EJB and call a method on another EJB if the expected result is not returned. The deployment

descriptor for this MBean defines how often the check is to occur, the type of the database access EJB and the attribute(s) that are to be checked, the expected value of the checked attribute(s), and the type and method to call of the application shutdown EJB. The MBean is parameterized with this deployment descriptor and deployed into the JMX server.

At application startup, the MBean locates and stores a reference to the database access EJB. The MBean is activated at the interval defined in the deployment descriptor to read the specified status attribute. If the attribute returns the correct value (i.e., accessible) the MBean sleeps for the appropriate amount of time. If the attribute returns an incorrect value (indicating that the database is not accessible) the MBean calls the appropriate method on the shutdown EJB to gracefully terminate the application.

4. Related Work

There has been extensive effort in identifying design approaches useful in high-confidence applications. However, most of this work has been applied to traditional development practices. Our focus is on whether and how one can use such proven techniques in component-based applications. The underlying premise is that mechanisms can be developed or tailored to provide a separation of HCS concerns from core application logic, presumably facilitating development.

Aspect-oriented programming (AOP) [4] and multi-dimensional separation of concerns [7] offer the separation of application "core classes" from non-functional, cross-cutting concerns (aspects). Automated mechanisms have been developed that support "weaving" the code that implements the aspects throughout an application, as well as integrating such aspects across multiple dimensions of composition. Our approach has a focus on the HCS concerns of the system, with a similar goal of separation of such concerns from component application logic, including mechanisms that support system composition in cases where the components include both developed and third party components. Our HCS mediators and monitors can be considered as an integration mechanism for the HCS aspects of a system, and, as they are based on extensions to a standard framework, should facilitate adoption.

Our HCS container approach is similar to the Object Infrastructure Framework (OIF) described in [1], which is an approach to achieve non-functional "ilities" through the use of "injectors" attached to communications. They have experimented with their approach by developing an implementation of the framework based on CORBA. Meta-information is "attached" to CORBA method invocation, identifying the sequence of injectors that is to be processed for that communication. Using this approach, they have provided support for a number of "ilities", including reliability, maintainability, quality of service and security. The OIF approach to attaching "injectors" to communications is similar to our interceptor-based mediators, except that our approach is encapsulated within a standard container that is configurable at deployment time. We recognize that for some HCS needs, the mediator approach is

not sufficient. For these, we have also developed the monitor capability, and demonstrated that it can be implemented in a manner to allow for a seamless integration with the mediator capability.

5. Summary/Future Directions

This paper describes our approach to extend standard container technology so that it can better support High Confidence Software. With CBS moving toward increased flexibility, as evidenced by the interest in dynamic reconfiguration, hot-swappable components, and independent extensibility, approaches are needed so that such flexibility can be effectively accommodated in a manner that supports predictable, robust behavior. With such approaches, one can realize the benefit of the flexibility without suffering from unpredictable behavior, a need that is fundamental to HCS.

Our approach to offering HCS services to component-based systems is based on mediators, which control component communications, and monitors, which periodically check application state. The mediators and monitors have been implemented in our system as augmentations to standard container and server technology. With these augmentations, we can support a variety of services of interest in HCS, including pre and post condition evaluation, watchdog monitoring, and policy enforcement. Incorporating such traditional HCS services in a standardized framework can allow for more effective implementation of HCS systems, enabling such systems to realize the benefits being offered by CBS.

Our focus is less on the development of a new approach to offer separation of concerns, but rather on the tailoring of existing technology to more effectively support HCS design techniques. We would like to see if, for example, we can achieve a benefit of a HCS safety kernel implemented via EJB-based HCS mediators and monitors similar to the e-commerce benefits realized from EJB container-based services supporting transactions. At the same time, we intend to investigate the some of the drawbacks of using such an approach for component based HCS. For example, some of the flexibility offered with a CBS approach is limited with this framework (e.g. for component monitoring, the components must be monitorable in a manner consistent with the framework). Also, for some uses a mediator-based approach may present a performance penalty, as there are additional interceptors being applied to each communication. Our prototype system will help us to evaluate the difficulties and benefits of this approach for HCS.

We plan to further refine our prototype, experimenting with additional types of mediators and monitors. We have limited our experiments to components that explicitly expose attributes, but in the JBoss environment the Java Reflection API could be used to access non-public attributes and methods. However, we expect that components in a high confidence composition would expose some sort of standard interface - for example, an interface that deals with component status. We plan to further investigate what characteristics are needed in such an interface.

6. References

[1] Filman, R., et. al., “Inserting Ilities by Controlling Communications.” Communications of the ACM, vol. 45, no. 1, January 2002.

[2] Hermann, D., Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors. IEEE Computer Society, 1999.

[3] High Confidence Software and Systems Coordinating Group, “High Confidence Software And Systems Research Needs.” Interagency Working Group on Information and Technology Research and Development, January 2001.

[4] Kiczales, G. et. al., “Aspect Oriented Programming”, Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[5] Leveson, N., Safeware: System Safety and Computers. Addison-Wesley, 1995.

[6] Meyer, B., “Applying Design by Contract". IEEE Computer, 25(10): 40-51, October 1992.

[7] Tarr, P., et. al., “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” Proceedings of the 21st International Conference on Software Engineering (ICSE), Los Angeles, May, 1999.