Containers for Predictable Behavior of Component-based Software

Gary J. Vecellio
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508
1+ 770 739-8598
vecellio@mitre.org

William M. Thomas
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508
1+ 703 883-6159
bthomas@mitre.org

Robert M. Sanders
The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508
1+ 703 883-7602
rsanders@mitre.org

ABSTRACT

Component developers have limited knowledge of how their components will be aggregated into applications and they can not control the deployment and execution environment. This makes the development of predictable component-based software a difficult proposition. Adding services to a software container can help remedy this problem. This paper discusses how commercial container technology can be augmented to support more predictable behavior of component compositions. Our approach consists of augmenting an open source Enterprise JavaBeans™ (EJB™) container and server with assertion capabilities. We discuss how these new capabilities can be used at load and initialization time to verify that a composition meets some policy constraints and at runtime to verify that the composition is maintaining critical properties.

Keywords

Assurance, Security, Components, Containers, Frameworks, Java, Enterprise JavaBeans, Java 2 Enterprise Edition

1. INTRODUCTION

A high confidence system is one that behaves in a well-understood and predictable fashion and where the consequences of failure are high. It should withstand attacks as well as naturally occurring hazards, and must not cause or contribute to accidents or unacceptable losses. High Confidence Software (HCS) is needed if high confidence systems are going to meet their requirements. This means the development of safe, reliable, dependable, secure, and survivable computing and communications software.

There are fairly well established techniques and processes for developing HCS. Techniques include Design by Contract™, N-version programming, watchdog timers, and recovery blocks. Process-related standards for HCS include DO-178-B, and MIL-STD-882-C. When applied to systems with high confidence requirements (e.g., weapon release and aircraft control software), these techniques and processes have resulted in successes. However, these techniques and processes are generally expensive to employ, and are not tailored to efficiently support component-based software development.

For the purpose of this paper, we will define a *component* as a reusable aggregation of functionality with a well-defined interface and a *container* as software that supports the execution of components. Assuming current practices, components and containers are generally developed by different individuals or organizations. Component developers are experienced in, for example, a business domain while container developers are experienced in system level programming. This separation of concerns allows developers to utilize their strengths and to reuse the strengths of others, i.e., allowing component developers to inexpensively utilize services that they would otherwise have to develop.

In the Java™ 2 Platform Enterprise Edition (J2EE™) framework, container-provided services include: transactions, security, connection pooling, object passivation, and others. The goal of J2EE is to support the quick and efficient development of component–based e-business applications.

While container services offered by J2EE map well to the e-business problem space, they do not offer the services appropriate for building HCS. However, in the same way J2EE decreases the cost of e-commerce software, augmenting containers to support HCS techniques may decrease the cost of HCS development. For example, supporting these techniques in the container makes the application easier to modify and maintain. That is, it is easier to modify a missile release component if the missile release interlocks are implemented, verified, and asserted separately, in the container.

One approach to adding HCS mechanisms could be to design a new container technology. Another approach could be to augment an existing container technology. We chose to demonstrate our approach by augmenting an Enterprise JavaBeans™ (EJB™) container that is part of the open source JBoss project[1]. Our reasoning was that basing our work on an already established container technology would ease adoption. Also, using an established container technology might support the deployment of applications with mixed levels of assurance. Finally, using a preexisting container technology would allow us to concentrate on the novel aspects of our work and reuse much of the existing container design and implementation.

2. ASSURANCE THROUGH ASSERTIONS

When creating a software system from components the composer needs to consider the relationship among the components. Usually this relationship is one of client and service provider. However, as correct behavior becomes more critical, the other properties among the components may become important. For example, the execution of a critical service might only be acceptable if the other components in the system are in a valid state. Or service execution might only be acceptable if all of the

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

deployed components offering the same service have arrived at the same result. Because these properties are established over the set of deployed components they cannot be addressed in the individual components, but must be collectively addressed at composition, deployment, or at runtime.

In their typical software engineering usage, assertions provide a means for specifying expectations about a module's implementation. Assertions are typically a Boolean-valued expression that, if evaluated to true, have no effect, and if evaluated to false, result in an error being reported. For HCS, verifying that assertions are met at runtime provides additional confidence that the implementation is consistent with the intent of the developer, composer, and deployer.

Meyer introduced Design by Contract™ (DBC) as a means to specify constraints on the design and use of a module. The contract applies to both the implementation of a module and the use of a module [3]. DBC emphasizes that it is important not only to determine that the components comprising the system are implemented in a manner in accordance with their own development expectations, but also that they are being used in a manner in accordance with the constraints laid out by the module developer. When we investigated the application of DBC techniques to CBS we concluded that improvements to the techniques and mechanisms might make container-based composition easier.

In a software composition setting, it is essential 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. Since the components of the system may have been purchased (rather than developed), the composer may not have the ability to modify the individual components. To better support assurance in composed systems, we need mechanisms for providing assertions on the compositions of components.

Assertions can also be used to describe and enforce properties in addition to the DBC properties discussed above. For example, many of the design techniques that are used to develop high confidence software can, at least in part, be implemented by using assertion like mechanisms. Three examples of high confidence design techniques are:

- Software interlocks – verifying all preconditions for critical operations are correct. Uses assertions to describe the preconditions of critical operations.
- Watchdog timers – verifying at predetermined intervals that a component set is in a consistent state. Uses assertions to describe the state invariants for the components in an application.
- Software firewalls – verifying that in the presence of an error, clients will receive a predetermined notification. Uses assertions to describe the postconditions of operations.

Consider a scenario in which we wish to execute a particular function of a critical component based upon the status of a set of dependent components. Lets assume that the critical component controls the launch of a surface to air missile (Fire) and the dependent components determine the nationality of the target (Identify Friend or Foe), if the target is closing on allied forces (Threat), and the status of the missile launcher is (Launcher Ready). Figure 1 represents the hypothetical missile firing system. Let's say one of the simpler missile release interlocks states that all software components must be in an error free state. That is, ∑ IFF (State OK) AND ∑Threat (State OK) AND Launcher (State OK) == TRUE.

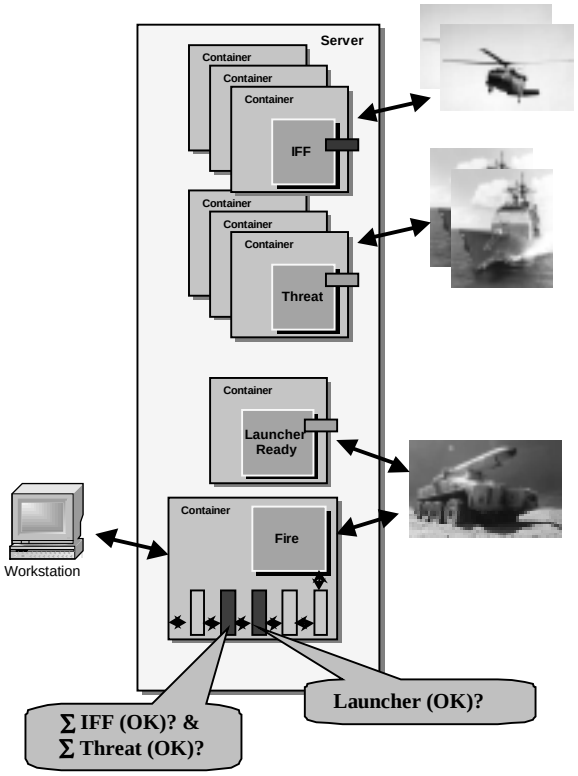

Figure 1: Example of a runtime assertion that verifies all Identify Friend or Foe components, all Threat components, and the Launcher Ready component are in a valid state before the Fire component will execute.

Since this interlock is dependent on the components deployed in the system it cannot be implemented without knowing the deployment configuration. But this causes component – deployment configuration coupling, something that should be avoided. A solution to this issue is to have the container verify the interlock. As shown in the figure the fire component container asserts that all components are in a valid state. This decouples the interlock from the missile fire component and allows verification of the interlock to occur at the container level. This means that a fire component replacement will observe this interlock, which will minimize re-verification cost, improve the rate at which new systems can be deployed, and support runtime replacement of components.

3. HCS CONTAINERS

For our experimental prototype we chose EJB as our component and container technology for two reasons. First, it is widely used in industry making the adoption of our augmentations easier. Second, it is based on two "design patterns" that fit our goals. These "design patterns" are:

- Indirection – the EJB container mediates interaction with EJB components.

- Declarative programming – the behavior of the EJB can be changed without source code modifications (The JBoss implementation also supports declarative programming for the server and EJB containers).

These "design patterns" are the basis of our research into HCS containers. That is, we use container level indirection to verify high confidence properties before and after a method is executed. At deployment time we use declarative programming to adapt which properties are verified. Deployment time adaptations can be used to adapt components to varying configurations or deployment environments. In addition, the JBoss implementation allows the insertion of monitors into the EJB server. These monitors are peers of the EJB containers. They have access to all containers and EJBs loaded in the server.

We are investigating two types of EJB server and container enhancements: mediators, which control communications among components and between a client and server; and monitors, which evaluate assertions outside the context of component communication.

Figure 2 is a high level view of the JBoss Server Architecture. JBoss server is based on a Java™ Management Extensions (JMX) framework. The server's EJB containers are built up dynamically and loaded into the JMX framework. The EJB containers are based on another 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.

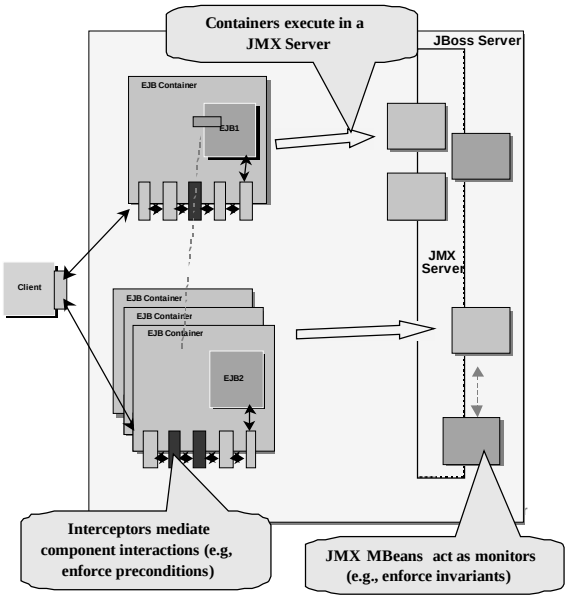

Figure 2: Overview of the JBoss architecture

In the JBoss Container Framework, plug-ins implement specific interfaces and are selected by specifying them in an XML-based 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 interceptors 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 bean level (i.e., defining a new container type that is local to a specific EJB). We also 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.

Interceptors can be used to implement precondition, postcondition, and invariant assertions described in DBC. Preconditions are evaluated before an EJB method is called and the invariants and postconditions are evaluated before the results are returned to the client. This is slightly different than DBC because the container rather than the client evaluates the preconditions. We see this as an advantage because it means the client is required to have less knowledge about the service.

Monitors can be used to evaluate application level invariant assertions. These evaluations can occur at timed intervals or an interceptor can kick them off when a method invocation occurs.

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.

4. SAMPLE PROBLEMS

We have developed extensions to the JBoss container that support CBS assertion checking. While our prototype implementation is specific to the JBoss implementation, the concepts should apply to other EJB containers. Our extensions include definition of new EJB container types, modifications to the EJB deployment descriptor and server configuration files, development of new interceptors, and development of new JMX Mbeans.

4.1 Software Interlocks

Consider the simple example from section 2. That example showed checking of dependent component status before method execution was allowed. It was an example of a simple software interlock. Some might argue that, for HCS, checking dependent component status should occur before all method invocations.

In this section we present a slightly more involved example of how a software interlock could be implemented. These interlocks can establish the preconditions for executing the critical methods of an EJB. Suppose we have a Weapon Control EJB that implements a method (Fire) that will release the weapon. We have an Identify Friend or Foe (IFF) EJB that determines the nationality of a target. We have a threat evaluation EJB that determines if the target represents a threat to allied forces (for example closing on an allied position). Finally, we have a Launcher EJB that determines the weapon and launcher status. For safety reasons we wish to prevent the firing of the weapon in all cases except when:

- all IFF sensors identify that the target is Hostile (as opposed to Friendly, or Unknown),
- all Threat evaluation components identify that the target is Closing (as opposed to Not Closing, or Unknown), and

- the Launcher is Ready (as opposed to Not Ready, or Error).

When all three assertions are satisfied, the weapon is allowed to fire.

This interlock implementation decouples the safety interlocks from the business logic implemented in the IFF, Threat Evaluation, Launcher, and Weapon Control EJBs. The safety interlocks are implemented as interceptors in the Weapon Control container. The Weapon Control container is parameterized at startup by the Weapon Control deployment descriptor.

The interceptors receive control when the EJB is first loaded, just before the application begins execution, and when the application is shut down. In this example, when the Weapon Control containers interceptors receive control just before the application begins execution they check the application environment for containers that contain the dependent EJBs. That is, the application is checked for IFF, Threat Evaluation, and Launcher containers. References to these containers are stored in the interceptors for later use.

The deployment descriptor for the Weapon Control EJB will specify the Java classes that are to be used for the interceptors and the dependent EJB container types for each interceptor. In addition, the deployment descriptor could be used to specify the dependent EJB attribute that is checked and the expected result of evaluating the assertion.

On a call to the Weapon Control EJB, the interceptor chain will be activated. The processing of the chain is as follows:

- 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.
- Similarly, the second 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 Threat EJB(s) if the target is closing. If all agree that the target is closing, control will pass to the next interceptor in the chain. Otherwise, an exception is returned to the client.
- Finally, the third 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 Launcher EJB(s) if the launcher is ready. If the launcher is ready, control will pass to the next interceptor in the chain (in this example the next interceptor is not concerned with the interlocks). Otherwise, an exception is returned to the client.

The interlock interceptor chain is transparent to callers of the fire method and is decoupled from the Weapon Control EJB. Modification to the interlocks can be done by changes to the deployment descriptor or the interceptors, without impacting the client code or the Weapon Control or dependent EJB. Deployment of a different Weapon Control EJB in the same Weapon Control container allows the new Weapon Control EJB to be controlled by the original interlocks.

4.2 Watchdog Timers

The software interlock example above used interceptors to establish and enforce preconditions on the execution of critical methods of an EJB. In this example we will describe the use of JMX MBeans to establish and enforce application wide invariants.

Suppose we have an application that contains one or more components that use services outside the control of the application. For example, the application may depend on a database that is located at a different location. To function properly this application must be able to connect to and exchange information with the remote database. The fact that this application is dependent on a remote database is not 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. Other components in the application might not know that they have to account for the database component returning an error when it cannot access to database.

This application can use an application wide invariant to enforce that the remote database is always accessible. This invariant can be enforced as follows. Develop a JMX MBean that will periodically check the status attribute of an arbitrary EJB and call a method on another EJB if the correct result is not returned. A deployment descriptor for this MBean defines how often the check is to occur, the type of the database access EJB, the attribute to check on the database access EJB, the expected value of the checked attribute, the type of the application shutdown EJB, and the method to call on the shutdown EJB. The MBean is parameterized with the deployment descriptor and deployed into the JMX server.

At application startup, the MBean locates and stores a reference to the database access EJB. At an interval defined in the deployment descriptor the MBean is activated and it accesses the attribute defined in the deployment descriptor. If the attribute returns the correct value the MBean sleeps for appropriate amount of time. If the attribute returns an incorrect value the MBean calls the appropriate method on the shutdown EJB to gracefully terminate the application. This example shows how the JMX framework can be used to enforce application wide invariants.

4.3 Software Firewalls

Software firewalls are used to protect or wall off portions of an application. For purposes of this example, we assume we want to place a firewall between client code that uses an EJB and the EJB that executes in a server. We also assume the purpose of the firewall is to prevent the propagation of Java runtime exceptions to the client. Firewalls could also be placed between any two EJB and could also be used to prevent the return of erroneous values.

Interceptors are used to implement software firewalls as follows. Develop an interceptor that passes the invocation unmodified when a method is called and maps any runtime exceptions propagated by the method to an exception type that is defined in the deployment descriptor. Rather than a many to one mapping the deployment descriptor could also specify a more complex mapping among exception types. For example, if a component propagates a java.lang.RuntimeException or some subtype of java.lang.RuntimeException the interceptor catches the exception and throws org.mitre.fatalServerError, also a subtype of java.lang.RuntimeException.

This rather simple example shows a container can alter the externally visible behavior of a component. Alternately, this example shows how a container can force independently developed components to adhere to a predefined client/server exception handling policy. That is, clients will only see one type

of java.lang.RuntimeException regardless of the type thrown by the component.

4.4 Composition Policy Enforcement

The previous examples illustrate the use of mediators and monitors for the runtime checking of composed applications. Our approach similarly provides the capability for load and initialization time checking of system wide policies.

Consider a multi-sensor targeting system like that described in section 4.1. Assume this application has a policy that states there must be at least two IFF sensors available before the system can transition to an operational state. Enforcement of this policy can be accomplished with an interceptor that checks the application policy when the interceptor receives notification that the application is about to start.

Similarly, an MBean could be notified when an application is loaded and the MBean could carry out the policy enforcement. For example, the compositional policy for an application could state that only EJBs that carry third party certification can participate in the application. At application load time the MBean could check each .jar file in the application for the digital signature of a third party certification authority. The MBean would unload the application if the digital signatures were not present or were incorrect.

5. RELATED WORK

Our approach is similar in intent with work in the areas of aspect-oriented programming (AOP) [2] and multi-dimensional separation of concerns [4]. AOP focuses on separation of the application's "core classes" from the non-functional cross-cutting concerns (aspects) and offers a development mechanism to support weaving the code supporting the different aspects throughout an application. The work on multi-dimensional separation of concerns described in [4] identifies the need for separation of overlapping concerns along multiple dimensions of composition and decomposition. They define "hyperslices" and "hypermodules" as flexible mechanisms to support decomposition and composition. The hyperslices are written to encapsulate each dimension of concern, and are integrated via hypermodules to form the completed system. Our approach is based on similar goals of separation of HCS concerns from component application logic, and enabling application level assurance and verification in a system composed with third-party components. Our HCS container can be considered as an integration mechanism for the HCS aspects of a system, but a mechanism that is based on small extensions to the EJB model.

The Object Infrastructure Framework (OIF) described in [1] is a similar approach focused on achieving non-functional "ilities" via "injectors" attached to communications. They have experimented with their approach by developing a CORBA-based implementation of the framework, attaching meta-information to CORBA method invocations which identifies the sequence of injectors that are to be processed for that communication. The "ilities" that have been addressed with this approach include reliability, maintainability, quality of service and security. The OIF approach to intercepting communications is similar to our interceptor-based approach, except that our approach is encapsulated within a container that is configurable at deployment time.

The SEI is investigating the use of prediction-enabled component technology (PECT), integrating component technology with analysis technologies [5]. This integration allows analysis and prediction of assembly-level properties prior to assembly. Our investigation of mechanisms to determine at deployment time satisfaction of state properties may be considered as a use of such prediction capability. For HCS, we are interested in providing assurance in addition to deployment-time prediction; for those needs we also support runtime techniques to monitor preservation of the properties. With these techniques we support assertions on the collective behavior of a component assembly that is specified and enabled outside of the context of the individual components. This supports the higher level of composition contracts needed to utilize CBS in high-integrity systems.

With respect to the goals of this workshop on predictable assembly, a fundamental goal of HCS is achieving predictable software. CBS presents some challenges for predictable behavior in that the runtime context of the components is not known at development time. Our HCS containers provide a mechanism to constrain component behaviors both at deployment time and runtime to better ensure that behavior is within the expected bounds. At deployment time, an EJB deployment can be rejected if an assembly does not satisfy the deployment assertions. This would constrain the application from starting in an unexpected state. At runtime, assertions can be monitored such that if an unexpected state is detected, the application can be transitioned to a different, presumably safer state. Such an approach can provide more confidence that the system is in a known state even in cases where the actual component behavior differs from its predicted behavior, which is an essential design goal of HCS.

6. CONCLUSIONS/FUTURE DIRECTIONS

Currently we are experimenting with a prototype system that supports a set of augmentations to the JBoss EJB container. Via these augmentations, we can support some straightforward evaluation of assertions at deployment time and at runtime. Our initial focus has been on the development of interceptors to support communication mediation. We also plan to develop MBeans to support periodic monitoring of state correctness, providing additional monitoring and recovery support to help an application ensure state correctness. As an outcome of this project we hope to:

- Develop a better understanding as to what HCS services would be of most benefit in a CBS context.
- Determine to what extent these services may be incorporated in standard container technology.
- Identify how verification of components and applications may be improved via container-based implementation of these services.

7. REFERENCES

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

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

[3] Meyer, B., "Applying design by contract". IEEE Computer, 25(10): 40-51, October 1992.

[4] 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.

[5] Wallnau, K., et. al., “Packaging Predictable Assembly with Prediction-Enabled Component Technology”, Technical Report CMU/SEI-2001-TR-024, Software Engineering Institute, November, 2001.