A Behavioral Model for Software Containers

Nigamanth Sridhar[1] and Jason O. Hallstrom[2]

[1] Electrical & Computer Engg., Cleveland State University
n.sridhar1@csuohio.edu
[2] Computer Science, Clemson University
jasonoh@cs.clemson.edu

Abstract. Software containers present an effective mechanism for decoupling cross-cutting concerns in software. System-wide concerns such as persistence, transaction management, security, fault masking, etc., are implemented as container services. While a lot of effort has been expended in developing effective container implementations, specifications for software containers are largely presented in informal natural language, which hampers predictable reasoning about the behavior of components deployed within containers. In this paper, we present a formal model for reasoning about the behavior of software containers. Our model allows developers to reason precisely about how the behaviors of software components deployed within a container are modified by the container. We further present the specifications of a few examples of container services that are found in different container implementations, and use our formal model to prove the correctness of the behavioral transformations that these services cause.

1 Introduction

A *software container* is a hosting environment for software components. It provides execution support to the components it hosts in a way that is similar to an operating system hosting processes. It also serves as a protective barrier, monitoring the interactions between hosted components and their clients, restricting the interactions to those that are deemed safe. Container-based models provide a clear separation of concerns between application logic and *enterprise services*, such as transaction management, persistence, security, etc.

Although there are several commercial container architectures, the best-recognized is the *J2EE* container provided by Sun Microsystems. The container provides a collection of enterprise services to its hosted components, selected from a predetermined set. This is not, however, the only service model to consider. Other models allow extensible service sets; some allow the service selection to change dynamically on a per-component basis [6]. We discuss some of these models in Section 2.

The services provided by a container affect *behavioral transformations* in the components it hosts. Metaphorically, the container is a type of lens: the clients' view of hosted components is altered by the services the container provides. Under this view, neither the client nor the component implementer needs to worry

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 139–154, 2006.

about implementing common enterprise services. Those services are provided by *every* component by virtue of being deployed within the container.

But there is a downside. When a component is hosted within a container, its specification no longer reflects its *total* behavior. Component clients must also consider the effects of the container services in their reasoning processes. The problem, however, is that existing container service descriptions are informal, and do not directly support formal reasoning. This is one of the key reasoning problems identified by the component-based software engineering community [5]. In the absence of more rigorous specification and reasoning techniques, the reliability of the container model may be undermined. This is the problem we address in this paper. Our contributions are:

1. A formal behavioral model of software containers, applicable across a range of commercial architectures.
2. Techniques for reasoning about the behavior of software components in the context of container-based deployment.
3. Examples of container service specifications from popular architectures, and a discussion how to use those specifications in the reasoning process.

The rest of the paper is organized as follows. In § 2, we present an overview of some existing container models. In § 3, we describe the different types of container services that can be described using our model. In § 4, we define a formal model that can be used to reason about the behavior of software containers and the services they provide. In § 5, we present some examples of services that can be specified using our model. After discussing related work in § 6, we conclude with a summary of our contributions and directions for future research in § 7.

2 Container Implementations

Software containers have been embraced as a means of modularizing cross-cutting concerns for a number of years. A container neatly encapsulates the services that cross-cut the components it hosts. Component developers need only worry about core functional concerns. In recent years, considerable effort has been invested in creating effective container implementations. We briefly examine some representative implementations.

EJB Containers. The Enterprise Java Beans (EJB) container [17] is the core of the Java 2 Enterprise Edition (J2EE) [18]. This architecture targets *enterprise class* systems. Applications are composed of EJB components that implement the business logic. The hosting EJB container provides transaction management, security services, etc. Although the component developer must abide by certain design constraints, he/she is shielded from the complexity of the underlying service implementations; the services are *container-managed*. Unfortunately, the container services are specified informally, precluding formal reasoning efforts.

Spontaneous Containers. In dynamic systems, such as those running in mobile networks, static service selection is insufficient. Popovici *et al.* [12] propose

spontaneous containers, combining container technology and dynamic aspect-oriented programming. In this model, the container is designed for application environments in which mobility and dynamic adaptation are required. A component that joins a network *discovers* available container services, and attaches itself to the appropriate services. The services are modeled as aspects that are dynamically woven into hosted components [13]. Further, the available service set can be extended dynamically. The system supports uniform service reconfiguration across hosted components without re-compilation or re-deployment. Again, formal reasoning is thwarted by a lack of formal documentation.

DRSS. Hallstrom et al. [6] describe *DRSS*, an open container architecture that allows for service variation across hosted components. The services supplied to a particular component can be applied and removed dynamically, allowing for runtime maintenance and evolution. The model is based on the notion of an *interceptor*, which processes messages flowing between component instances. Each invocation flows through an *interceptor chain*, before reaching the target component instance. Container services are implemented in the form of interceptors, making it possible for the container to inject services *between* component invocations. Since interceptor chains are allocated on a per component basis, all components need not use the same services. DRSS is also *spontaneous*, in the sense that it allows for dynamic discovery and deployment of services. Again, however, DRSS services are documented with informal descriptions alone.

As we will see, our reasoning framework applies to all these container implementations (and others), despite the variation that occurs across the models.

3 Classifying Container Services

In our formal model, we consider three classes of container services. We note that this classification is not exhaustive. For example, our model does not consider container services that ignore the specifications of hosted components. Our focus is to stay within the confines of contract-based reasoning.

1. **Monotonic addition of behavior.** This class of services are those that do not in any way modify the existing behavior of target components. They simply add functionality over and above what the components provide. Examples include *Message Logging* and *Object Visualization*.
2. **Redirection.** A service in this class remains faithful to the original specification of each hosted component C, but may redirect method invocations to an object different from the intended receiver. When a method call to a receiver object O_1 is redirected to a different object O_2, the service must meet the obligation that the new target O_2 is of the same type (or behavioral subtype) as that of O_1. Examples include *Fault Masking* and *Load Balancing*.
3. **Deferred Execution.** This class of container services may delay the delivery of a message to an object. The client may not immediately see the effect of a message sent to an object, but will see the effect at a later time. Examples include *Transaction Management* and *Batch Processing*.

4 Modeling Containers

4.1 The Need for a Formal Model

While a lot of work has gone into crafting different implementation strategies [2, 6, 7, 11, 12, 17, 18, 19], there has been relatively little effort focused on defining effective methods for reasoning formally about the behavior of components and systems that use these containers [5].

A software container acts as a mediator between a hosted component and its clients. Each message sent by a client to a hosted component is "seen" first by the container, which then sends the message to the component. Before the component receives the message, therefore, the container may have performed some actions. It may even have modified the message. Similarly, after the component has acted upon the client's message, its response to the client goes first to the container before reaching the client.

It is this "power" of the container that necessitates the need for a formal specification of its behavior. A contract that describes the behavior of a component may be useless if the component is instantiated within a container without a specification of *what* exactly the container is doing to its interactions. The specification must include details of how the specifications of hosted components are (or may be) modified by the container. Informal descriptions are not enough to achieve predictable correctness of software in the presence of containers.

4.2 The Behavioral Model

We begin our discussion of the formal model by looking at how a client may be affected by a container. When a client program makes a method invocation on a particular object, the rules of design-by-contract [10] tell us that the client must have satisfied the pre-condition of the method; and consequently, upon return from the execution of the method, the client will be able to assume that the post-condition is satisfied. This contract must not be compromised by the presence of a container. Consider a component $\mathcal{H}$ with method f_1:

$$\begin{array}{l} \textbf{Component} \quad \mathcal{H} \\ \quad \textbf{operation} \quad f_1 \\ \qquad \textbf{pre} - \textbf{condition} : \mathcal{P}_{pre} \\ \qquad \textbf{post} - \textbf{condition} : \mathcal{P}_{post} \end{array} \tag{1}$$

Further, let us suppose that this component is hosted in a container $\mathcal{C}$. When a client wants to use this component, it's view is now altered. The component that the client is using is $\mathcal{H}[\mathcal{C}] = \mathcal{C} \oplus \mathcal{H}$. We use the operator $\oplus : \mathit{Container} \times \mathit{Component} \rightarrow \mathit{Component}$ to denote the composition of a component with a software container. We use $\mathcal{H}[\mathcal{C}]$ to denote that component $\mathcal{H}$ is deployed inside container $\mathcal{C}$, and $h[\mathcal{C}]$ to denote an instance of $\mathcal{H}$ deployed in $\mathcal{C}$.

In accordance with the principles of modularity, the specification presented above must hold regardless of the context in which f_1 is invoked. The client, after the call to f_1 must be able to assume $\mathcal{P}_{post}$. Consequently, even if the component $\mathcal{H}$ is hosted in a container, this post-condition must still hold for $\mathcal{H}[\mathcal{C}]$; the

modified post-condition must be at least as strong as the original. Similarly, the client will only have to worry about satisfying the original pre-condition. If the container were to change the pre-condition, it can only be weakened.

The statement above then views a container-hosted component as a *behavioral subtype* [9] of the "*bare*" component. The container-hosted component, therefore, honors the original contract. However, it does more, and this additional behavior is *not* captured by the contract. While we can state that the new pre-condition (post-condition) is weaker (stronger) than the original, we do not have a notion of *exactly how much weaker (stronger).* Behavioral subtyping is therefore not enough for our purpose.

Model of a Container Service. Each container service $\mathcal{CS}$ contains:

- Zero or more state variables that the service may use. These variables define what the service does, and its effect on each method invocation.
- A predicate $\mathcal{M}_{pre}$ that specifies how the service modifies the pre-conditions of target methods. We call this the **pre-modifier**.
- A predicate $\mathcal{M}_{post}$ that specifies how the service modifies the post-conditions of target methods. We call this the **post-modifier**.
- The body of the service: the actions that define the functioning of the service.
- A set of additional methods that a client may invoke on a hosted component; we can view these methods as being added to the component's interface.

The minimal structure of the specification of a container service is as follows:

Definition 1. *Container Service*

Service $\mathcal{CS}$
State : $\mathcal{S}$
Pre_modifier : $\mathcal{M}_{pre}$
Post_modifier : $\mathcal{M}_{post}$
Body : $a_1; a_2, \ldots, a_n$
Methods : $m_1, m_2, \ldots, m_k$
end Service $\mathcal{CS}$

The predicates $\mathcal{M}_{pre}$ and $\mathcal{M}_{post}$ are defined in terms of the service state variables ($\mathcal{S}$) and the *context* of each method call that the service is applied to. The context of a method call includes the name and signature of the method, the target object, and the values of all parameters to the method (if any). Upon return from a method, the context includes the current values of all the parameters, and the method's return value (if any). The methods $m_1, \ldots, m_k$ operate on the same set of variables. Encapsulation is respected; the service cannot access private fields or private methods of the target. In addition to its local state, each service has access to the following keywords:

- thismethod is a handle to the method to which the service is being applied.
- thismethod.args is the ordered sequence of parameters to thismethod. The sequence holds the input values of the parameters when referenced in

$\mathcal{M}_{pre}$, and the output values when referenced in $\mathcal{M}_{post}$. Additionally, #thismethod.args holds the input values of the parameters to thismethod in $\mathcal{M}_{post}$.
- thismethod.retval is the value returned by the method.
- target is the object that is intended as the receiver of the method call.

Each component service is *locally certifiable*. This means that we require each component service to be defined modularly and in isolation; the specification of the service is only dependent on its own local state, and is not modified by changes in other services in the container.

Model of a Service Group. Container services are aggregated into *service groups*, each of which is a sequence of container services. Components do not subscribe to a service directly; they subscribe to the service group that contains exactly the service(s) that are needed. This simplifies our reasoning model to consider only one kind of composition, without reducing expressivity.

Each service group in a container consists of the following:

- The string of container services that the service group collects. This string is referred to using the keyword services.
- The state of the service group, which is simply the fully qualified union of the states of the individual services in the service group. By fully qualified we mean that variables are of the form service-name.variable. If the same variable name is used by two services, the two variables are treated as distinct.
- A predicate $\mathcal{M}_{\mathcal{SG}\_pre}$ that specifies how the pre-conditions of target methods are modified by the service group. This predicate is the conjunction of the **pre-modifier**s of all the container services in services.
- A predicate $\mathcal{M}_{\mathcal{SG}\_post}$ that specifies how the post-conditions of target methods are modified by the service group. This predicate is the conjunction of the **post-modifier**s of all the container services in services.

Each service group is a sequence of services; the order in which the services execute is important. How then is it sufficient that the pre-modifier (post-modifier) of the service group is simply the conjunction of pre-modifiers (post-modifiers) of the individual services? Why do we not use the Hoare logic rule of sequential composition? Consider a service group $\mathcal{SG}_k$ with three services $\mathcal{CS}_1$, $\mathcal{CS}_2$, and $\mathcal{CS}_3$ with pre- and post-modifiers as follows:

$$\begin{aligned} \mathcal{CS}_1 &:: \langle \mathcal{M}_{1\_pre}, \mathcal{M}_{1\_post} \rangle \\ \mathcal{CS}_2 &:: \langle \mathcal{M}_{2\_pre}, \mathcal{M}_{2\_post} \rangle \\ \mathcal{CS}_3 &:: \langle \mathcal{M}_{3\_pre}, \mathcal{M}_{3\_post} \rangle \end{aligned} \tag{2}$$

According to the rule of sequential composition, if we had

$$\mathcal{M}_{1\_post} \Rightarrow \mathcal{M}_{2\_pre} \wedge \mathcal{M}_{2\_post} \Rightarrow \mathcal{M}_{3\_pre} \tag{3}$$

then we can say that the following about the pre- and post-modifier of $\mathcal{SG}_k$:

$$\mathcal{SG}_k :: \langle \mathcal{M}_{1\_pre}, \mathcal{M}_{3\_post} \rangle \tag{4}$$

However, since our services are all defined independently, we cannot assume such relationships as in (3) above. If $\mathcal{CS}_1$ does nothing to ensure $\mathcal{M}_{2\_pre}$ and $\mathcal{M}_{3\_pre}$, then these two predicates must be true *before* $\mathcal{CS}_1$ executes. Similarly, if $\mathcal{CS}_2$ and $\mathcal{CS}_3$ do not do anything to disrupt $\mathcal{M}_{1\_post}$, then this predicate still holds at the end of $\mathcal{CS}_3$. Hence we cannot use the rule of sequential composition to determine the pre- and post-modifier of a service group.

The formal definition of a service group is as follows. Note that the function *elements: String* $\rightarrow$ *Set* returns the elements in a string as a set.

Definition 2. *Service Group*

Service Group $\mathcal{SG}$

Modeled by: *services* : *string of* $\mathcal{CS}$

State : $\mathcal{S}_{\mathcal{SG}} = \{\forall\, cs \in elements(\mathsf{services}) : \mathcal{S}(cs)\}$

pre − modifier : $\mathcal{M}_{\mathcal{SG}\_pre} = \bigwedge_{i=1}^{|\mathsf{services}|} \mathcal{M}_{i\_pre}$

post − modifier : $\mathcal{M}_{\mathcal{SG}\_post} = \bigwedge_{i=1}^{|\mathsf{services}|} \mathcal{M}_{i\_post}$

methods : $\mathsf{methods}(\mathcal{SG}) = \{\forall\, cs \in elements(\mathsf{services}) : \mathsf{methods}(cs)\}$

When a component $\mathcal{H}$ subscribes to a service group $\mathcal{SG}$, $\mathcal{H}$ is transformed to include the state variables in $\mathcal{S}_{\mathcal{SG}}$, and all the methods in $\mathsf{methods}(\mathcal{SG})$. Moreover, the pre-condition (post-condition) of every method in $\mathcal{H}$ is transformed to include the pre-modifier (post-modifier) of $\mathcal{SG}$. For example, if the component defined in (1) subscribes to the service group in Definition 2, the component will be transformed as follows:

$$\begin{array}{l}\textbf{Component}\quad \mathcal{H}[\mathcal{SG}]\\ \quad\textbf{Additional State} : \mathcal{S}(\mathcal{SG})\\ \quad\textbf{operation}\quad f_1\\ \quad\quad\textbf{pre} - \textbf{condition} : \mathcal{P}_{pre} \wedge \mathcal{M}_{\mathcal{SG}\_pre}\\ \quad\quad\textbf{post} - \textbf{condition} : \mathcal{P}_{post} \wedge \mathcal{M}_{\mathcal{SG}\_post}\\ \quad\textbf{Additional methods:}\ \mathsf{methods}(\mathcal{SG})\end{array} \tag{5}$$

There is one more thing that we need to ensure before we can claim that this transformation is correct. Since the new pre-condition (post-condition) is the conjunction of the original pre-condition (post-condition) of the method f_1 with the pre-modifier (post-modifier) of $\mathcal{SG}$, we require the following to protect the conjunction, preventing the derivation of **false** statements:

$$\mathcal{M}_{\mathcal{SG}\_pre} \not\Rightarrow \textbf{false} \wedge \mathcal{M}_{\mathcal{SG}\_post} \not\Rightarrow \textbf{false} \tag{6}$$

We can now consider the rule required to prove that the transformation of a component by a service group is indeed correct.

Rule 1 (Behavioral Transformation)

$$\frac{\begin{array}{l}\mathcal{H} :: \langle h.f_1.\mathcal{P}_{pre}, h.f_1.\mathcal{P}_{post}\rangle \\ \{\mathcal{H}[\mathcal{SG}].f_1.\mathcal{P}_{pre}\}\ \mathcal{SG}.Body.pre\ \{\mathcal{H}.f_1.\mathcal{P}_{pre}\} \\ \{\mathcal{H}.f_1.\mathcal{P}_{post}\}\ \mathcal{SG}.Body.post\ \{\mathcal{H}[\mathcal{SG}].f_1.\mathcal{P}_{post}\}\end{array}}{\mathcal{H}[\mathcal{SG}] :: \langle h[\mathcal{SG}].f_1.\mathcal{P}_{pre}, h[\mathcal{SG}].f_1.\mathcal{P}_{post}\rangle}$$

The first antecedent requires that the specification of the method f_1 be established for the component $\mathcal{H}$ outside the container. The second antecedent requires us to show that the body of the service group when applied to the method call *on its way to* the target ($\mathcal{SG}.Body.pre$) meets the original method contract in terms of ensuring that the original pre-condition of the method f_1 holds at the end of the service execution. The third antecedent requires us to show that the body of the service group when applied to the method call *on its way back* to the client ($\mathcal{SG}.Body.post$) meets the original method contract in terms of ensuring that the original post-condition of the method f_1 holds at the end of the service execution.

Although the total number of service groups in a container is $\mid \Sigma^* \mid$ where Σ is the alphabet consisting of all the services provided by a container, some equivalences can be established among these groups. For any two service groups $\mathcal{SG}_i$ and $\mathcal{SG}_j$, if the set of services they include are the same, then the two groups are equivalent. The following is true of service groups in a container.

$$\begin{array}{l} elements(\mathcal{SG}_i.\mathsf{services}) = elements(\mathcal{SG}_j.\mathsf{services}) \quad \Rightarrow \\ \quad (\mathcal{S}(\mathcal{SG}_i) \equiv \mathcal{S}(\mathcal{SG}_j)) \wedge (\mathcal{M}_{\mathcal{SG}_i\_pre} \equiv \mathcal{M}_{\mathcal{SG}_j\_pre}) \wedge \\ \quad (\mathcal{M}_{\mathcal{SG}_i\_post} \equiv \mathcal{M}_{\mathcal{SG}_j\_post}) \wedge (\mathsf{methods}(\mathcal{SG}_i) \equiv \mathsf{methods}(\mathcal{SG}_j)) \end{array} \tag{7}$$

At any point during a method invocation, the service group is able to examine itself to determine which services have been applied so far. To do this, we use two trace variables. τ_{pre} denotes the trace of service invocations applied to a method call on its way from the caller to the callee, and τ_{post} is the trace of service invocations applied on its return from the method. Upon successful completion of a method call (when all services in $\mathcal{SG}$ have been applied, and the method has terminated), the following is true of the trace variables:

$$\begin{array}{l} \tau_{pre}(\mathcal{SG}) = \mathcal{SG}.\mathsf{services} \\ \tau_{post}(\mathcal{SG}) = reverse(\mathcal{SG}.\mathsf{services}) \end{array} \tag{8}$$

Model of a Software Container. With the pieces that we have established so far, we are now ready to describe the complete model of a software container. A software container $\mathcal{C}$ contains the following elements:

- Zero or more hosted components $\mathcal{H}_1$, ..., $\mathcal{H}_n$.
- Zero or more container services $\mathcal{CS}_1$, ..., $\mathcal{CS}_m$. These container services define the alphabet $\Sigma_{\mathcal{CS}}$ of the container $\mathcal{C}$.

– Zero or more service groups $\mathcal{SG}_1, \ldots, \mathcal{SG}_{|\Sigma^*_{\mathcal{CS}}|}$. The set of service groups is the set of all finite strings composed out of alphabet $\Sigma_{\mathcal{CS}}$.

As such, a container is modeled as a set of pairs, each mapping a hosted component to the service group that it subscribes to. $\mathbb{H}$ denotes the set of all components hosted in the container $\mathcal{C}$, and $\mathbb{SG}$ denotes the set of all service groups in the container. We model a software container, $\mathcal{C}$ as follows:

Definition 3. *Container*

$$\begin{aligned}\mathcal{C} = \{ \quad & \mathbb{H} = \{\mathcal{H}_1, ..., \mathcal{H}_n\}, \\ & \mathbb{SG} = \{\mathcal{SG}_1, \ldots, \mathcal{SG}_{|\Sigma^*_{\mathcal{CS}}|}\}, \\ & \langle \mathcal{H}_1, \mathcal{SG}_k\rangle, \langle \mathcal{H}_2, \mathcal{SG}_l\rangle, \ldots, \langle \mathcal{H}_n, \mathcal{SG}_m\rangle \quad \}\end{aligned}$$

Note: We first referred to a component $\mathcal{H}$ hosted by a container $\mathcal{C}$ as $\mathcal{H}[\mathcal{C}]$. Now, we refer to a component $\mathcal{H}$ subscribed to a service group $\mathcal{SG}$ as $\mathcal{H}[\mathcal{SG}]$. These refer to the same component — each hosted component must subscribe to exactly one service group in its hosting container.

5 Some Example Services

5.1 Message Logging

Logger (Fig. 1) is a container service that monotonically adds behavior to component methods to which the service is applied. The service does not cause any change in behavior to the original method. The service simply adds to the behavior of target methods by writing to a log the details of all calls.

The state of *Logger* consists of two strings — ILog and OLog. ILog is the log of all method invocations on their way from the caller to the target. Each element in the string is a pair consisting of a method name, and the sequence of actual parameter values passed to the named method. OLog is the log of all method invocations on their way from the target object back to the caller. OLog contains the final values of method parameters and the return value.

The pre-modifier of *Logger* adds to ILog a new pair with thismethod (method name), and the actual values of each element in thismethod.args (method arguments). @ILog here refers to the value of ILog in the state immediately preceding the start of the body of *Logger* during the target method call (*Logger.Body.pre*)[1]. The post-modifier of *Logger* creates a new pair with the name of thismethod and the sequence of return values (the actual values of elements in thismethod.args), concatenated with thismethod.retval. This new pair is added to OLog. @OLog here refers to the value of OLog in the state immediately preceding the start of *Logger* during the method's return.

Neither the pre-modifier ($\mathcal{M}^{pre}_{Logger}$) nor the post-modifier ($\mathcal{M}^{post}_{Logger}$) alter the pre- and post-conditional values of the target method's parameters. Thus, the

[1] We use the notation #x in the post-condition of a method to refer to the pre-conditional value of a variable x. The @x notation is used here to avoid confusion.

Service *Logger*
  State :
    ILog : *String of*
      ⟨ methodName : *String,*
        paramValues : *Sequence of parameter values*⟩
    OLog : *String of*
      ⟨ methodName : *String,*
        returnValues : *Sequence of return values*⟩
  pre_modifier : $\mathcal{M}^{pre}_{Logger}$
    ILog = @ILog ∗
      ⟨ thismethod.*name*,
        ⟨thismethod.args[0].*value*, . . . ,
        thismethod.args[| thismethod.args | −1].*value*⟩⟩
  post_modifier : $\mathcal{M}^{post}_{Logger}$
    OLog = @OLog ∗
      ⟨ thismethod.*name*,
        ⟨thismethod.args[0].*value*, . . . ,
        thismethod.args[| thismethod.args | −1].*value*⟩
          ∗ ⟨thismethod.retval⟩⟩
end Service *Logger*

Fig. 1. Specification of the Message Logging container service

second and third antecedents of **Rule 1** are true if the method is faithful to its behavioral specification (the first antecedent of **Rule 1**). Therefore, *Logger* causes a correct transformation.

5.2 Fault Masking

The next service we consider is one that causes the redirection of a method invocation to an object instance other than the intended receiver. In applications that provide fault tolerance, the fault masking service is very useful. The container can, in a manner that is transparent to the client, prevent method invocations from being sent to object instances that have failed. This kind of redirection only applies to components that are stateless—the method call cannot depend on the component's internal state.

The fault masking service needs the set of objects that are currently failed (R1), and for each failed object, the set of objects to which calls can be redirected (R2). There are different strategies for obtaining R1. In synchronous systems, simple timeouts can be used. In asynchronous systems, however, it is not possible to place such time bounds. We can, however, abstract away that

Service $FMask$

State :

fd : Failure detector oracle

suspects :$\{obj : obj \in \mathbb{H} : \mathsf{fd.failed}(obj)\}$

alt_objs :$Map(obj \rightarrow Set\ of\ obj)$

pre_modifier : $\mathcal{M}^{pre}_{FMask}$

target $\in$ suspects $\wedge$ alt_objs(target) $\neq \emptyset \quad \Rightarrow$

target = a_obj : a_obj $\in$ alt_objs(target) $\wedge$ a_obj $\notin$ suspects

post_modifier : $\mathcal{M}^{post}_{FMask}$ **true**

methods :

void setAlternates(a_objs: Set$<\mathfrak{T}$(target)$>$)

pre − condition :**true**

post − condition :

(target $\notin Keys$(alt_objs) $\Rightarrow$

alt_objs = #alt_objs $\cup$ $\{\langle$target, $a\_objs\rangle\}) \wedge$

(target $\in Keys$(alt_objs) $\Rightarrow$

alt_objs(target) = #alt_objs(target) $\cup$ $a\_objs) \wedge$

$(\forall o \in a\_objs : \mathbb{H} = \#\mathbb{H} \cup o)$

end Service $FMask$

Fig. 2. Specification of the Fault Masking container service

detail and leave it to some *failure detector* [3]. *FMask* (Fig. 2) maintains a set suspects—objects that the failure detector suspects to be failed. fd.failed(obj) is **true** for all suspects.

To satisfy R2, the service maintains a set of alternate objects (alt_objs) for every object *obj* in the container. This way, when the container does encounter a method call whose target is failed, it can look up an alternate object and forward the call to that object. The method setAlternates() can be invoked on a hosted object with a set of alternate objects. The argument to setAlternates() is a Set parameterized by the type of target. All objects in a_objs are added to the set $\mathbb{H}$ of container-hosted objects.

When *FMask* intercepts a method call to a suspected target, the call is directed to an object from alt_objs(target) that is still alive. If no such alternate object can be found by *FMask*, the invocation fails. On the return direction, the service does nothing, and therefore does not modify the post-condition.

Since neither $\mathcal{M}^{pre}_{FMask}$ nor $\mathcal{M}^{post}_{FMask}$ interfere with the pre- and post-condition of the target method, the second and third antecedents of **Rule 1** are true. Thus, if the original method meets its behavioral contract, *FMask* causes a correct transformation.

5.3 Transaction Management

We now come to an example of the third class of container services — deferred execution. The service we consider here is transaction management (Figure 3). Some components require that certain groups of methods be called in succession;

Service $TxnMgmt$

type Tx_Req_Types = $enum$\{Required, NotRequired\}

State :

tx_reqts :$Map(String \rightarrow$ Tx_Req_Types)

curr_txn : *String of* method

$\mu\tau$: *String of* method

pre_modifier : $\mathcal{M}^{pre}_{TxnMgmt}$

(tx_reqts(thismethod.$name$) = Required $\Rightarrow$

curr_txn = @curr_txn $*$ thismethod) $\wedge$ (target = $\perp$)

post_modifier : $\mathcal{M}^{post}_{TxnMgmt}$

(tx_reqts(thismethod.$name$) $\neq$ Required $\Rightarrow$

$\mu\tau = \#\mu\tau *$ thismethod) $\wedge$

(tx_reqts(thismethod.$name$) = Required $\Rightarrow$

expects commitTxn() $\vee$ rollbackTxn())

methods :

void setTxReqts(meth_name: String, tr: Tx_Req_Types)

pre − condition : **true**

post − condition :

(meth_name $\notin Keys(\#$tx_reqts) $\Rightarrow$

(tx_reqts = #tx_reqts $\cup$ $\{\langle$meth_name, tr$\rangle\}$)) $\wedge$

(meth_name $\in Keys(\#$tx_reqts) $\Rightarrow$

(tx_reqts(meth_name) = tr))

void commitTxn()

pre − condition : curr_txn $\neq <>$

post − condition :

curr_txn $=<>$ $\wedge$ ($\exists s$: *String of* method : $\mu\tau = s *$ curr_txn)

void rollbackTxn()

pre − condition : curr_txn $\neq <>$

post − condition :

curr_txn $=<>$ $\wedge$ ($\forall s$: *String of* method : $\mu\tau \neq s *$ curr_txn)

end Service $TxnMgmt$

Fig. 3. Specification of the Transaction Management container service

either all of these methods should succeed, or they should all fail. A partial execution may result in an inconsistent state.

A client using a component subscribed to the transaction management service must first identify the methods in the component that require transaction support. The state of *TxnMgmt* is defined in three parts:

tx_reqts: A map with the transaction requirement for each method in the component. Methods with no entry in tx_reqts do not need transaction support.

curr_txn: The string of methods that belong in the current transaction. Note that there can only be one "live" transaction at a time according to this specification; this is for simplicity of presentation due to lack of space. The specification can be extended to allow for multiple live transactions.

$\mu\tau$: The trace of all methods that target executes. This trace collects a method call when the method is actually executed, not when it is invoked. Therefore, for methods that do not need transaction support, the method is added to $\mu\tau$ upon invocation, since they are executed immediately. Methods that require transaction support are added to $\mu\tau$ when the transaction is committed.

The pre-modifier of *TxnMgmt* requires that the method being invoked be added to curr_txn if the method requires transaction support. If there is no live transaction when the target method is invoked (curr_txn $= \langle\rangle$), a new transaction is initiated. In addition, target is set to $\perp$, and control returns to the client.

The post-modifier $\mathcal{M}^{post}_{TxnMgmt}$ modifies the post-condition of the target to indicate whether it has been executed or not. The call is added to $\mu\tau$ if the method does not require transaction support. If the method does require transaction support, the method has not been executed yet; it has simply been entered into the current transaction. In this case, the post-modifier adds an **expects** clause [8] to the post-condition of the method; in the future, there has to be a call either to commitTxn() or to rollbackTxn().

In addition to the setTxReqts() method, *TxnMgmt* extends the interface of the target component with two more methods — commitTxn() and rollbackTxn(). commitTxn() can be called when there is a live transaction, and the method commits the transaction; all the methods in curr_txn are executed. This results in curr_txn showing up as a suffix of $\mu\tau$. Moreover, the post-conditional expectations of all the methods that belong in the current transaction are now met. The transaction can be rolled back using rollbackTxn(). This method simply throws away all the methods in curr_txn.

We now see how applying *TxnMgmt* to a component is a correct behavioral transformation. Assume that the correctness of the component's methods in isolation have been established (first antecedent in **Rule 1**). If a method does not require transaction support, the pre-modifier of the service does nothing to the method invocation. In case a method does require transaction support, it is not executed immediately. Instead, the *entire context* of the method is stored in curr_txn and the actual call is made when commitTxn() is called. When the method is eventually invoked, the state that the client called the method in originally is retained. In the case of a rolled-back transaction, the original target

method is never called. Thus, in all the above cases, the service does not affect the pre-condition. This proves the second antecedent in **Rule 1**.

The post-modifier of the service also does nothing in the case of methods that do not require transaction support. In the case of methods that do require transaction support, the client is required to call either commitTxn() (which will result in the post-conditions of all methods in the transaction being established) or rollbackTxn() (which will result in none of the methods being actually invoked). In all these cases, we have established the third antecedent in **Rule 1**. Thus, *TxnMgmt* causes a correct transformation.

6 Related Work

Our behavioral model of containers is related to the notion of behavioral subtyping [9]. The transformed component that the client sees as a result of the composition of the component with the container is a behavioral subtype of the original component. Our work goes beyond this, in that we are interested in specifying the additional behavior introduced by the container. Our work is loosely based upon the work on reasoning about object-oriented frameworks [14]. As in our case, Soundarajan and Fridella are interested in specifying application-specific behavior that results from specializing a framework. They use trace-based specifications to define how the template methods use hook methods to define application-specific behavior.

The work on modular aspect-oriented reasoning [4] shares similarities with our own. Clifton and Leavens show how modular reasoning techniques can be applied to understand the behavior of aspect advice. Their modifications to AspectJ in the form of *observers* and *assistants* provide a formal view of how exactly the behavior of a class is modified by the application of an aspect. The current work is derived from our previous work on modeling containers as parameterized components [16], where the container services are modeled as parameters. Our model of containers supports the dynamic addition and removal of parameters to a template. Our current approach to reasoning builds on previous work on dynamically bound parameterized components [15].

7 Conclusion

We began with the claim that the current state of container technology is an effective solution to the problem of cross-cutting concerns. Containers are used in a variety of scenarios to decouple system concerns from an application's core concerns. The software engineering community has been quick to respond to this growing demand for new and improved implementation strategies for software containers.

Although a good number of engineering issues with respect to container technology have been solved, the problem of predictable reasoning about software component behavior in the presence of software containers has not been well-studied. In this paper, we have presented a partial solution to this problem. We

have presented a formal behavioral model for software containers, along with the proof rule to show that an implementation of a service is, in fact, correct with respect to behavioral transformation.

The most important contribution of our model is the ability it affords to developers and users of software components that are deployed in a container environment to *accurately predict* the behavior resulting from composing a component with the container.

Our future plans include extending the model in ways that will allow for automated reasoning and verification of container services. We plan to build tools that will facilitate a combination of static verification and run-time monitoring to automate the reasoning process, at least partially. Our work in this direction involves extending the DRSS container architecture to include specifications. The specifications will be written in the Spec# [1] language, and will be checked at run-time to ensure that the services honor their contracts.

The work presented in this paper deals only with *behavioral* issues of container-component composition. Another direction for future research in this area involves extending our specification framework to specify *non-functional* properties of container services, such as performance, availability, etc.

References

1. M. Barnett, R. Leino, and W. Schulte. The spec# programming system: An overview. In *CASSIS 2004*, pages 49–69, 2005.
2. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software: Practice & Exp.*, (31):103–128, 2001.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
4. C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report TR-02-04a, CS, Iowa State U., 2002.
5. I. Crnkovic, H. Schmidt, J. A. Stafford, and K. C. Wallnau. 5th ICSE cbse workshop. In *Proc. 24th ICSE*, pages 655–656, Orlando, FL, May 2002.
6. J. O. Hallstrom, W. M. Leal, and A. Arora. Scalable evolution of highly-available systems. *IEICE/IEEE Joint Special Issue on Assurance Systems and Networks*, E86-D(10):2154–2166, October 2003.
7. JBoss. The JBoss home page. http://www.jboss.org.
8. S. Kumar, B. W. Weide, P. A. Sivilotti, N. Sridhar, J. O. Hallstrom, and S. M. Pike. Encapsulating concurrency as an approach to unification. In *FSE Workshop on Specification and Verification of Component-Based Systems*, October 2004.
9. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.
10. B. Meyer. *Design by contract*, chapter 1. Prentice Hall, 1992.
11. Object Management Group. CORBA Component Model, V3.0, 2002.
12. A. Popovici, G. Alonso, and T. Gross. Spontaneous container services. In *ECOOP 2003 – LNCS 2743*, pages 29–53. Springer, Aug 2003.
13. A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pages 141–147, New York, NY, USA, 2002. ACM Press.
14. N. Soundarajan and S. Fridella. Framework-based applications: From incremental development to incremental reasoning. In *Proceedings of the 6th International Conference on Software Reuse*, pages 100–116, London, UK, 2000. Springer-Verlag.

15. N. Sridhar. *Dynamically Reconfigurable Parameterized Components.* PhD thesis, The Ohio State University, 2004.
16. N. Sridhar and J. O. Hallstrom. Generating configurable containers for component-based software. In *6th ICSE Workshop on Component-Based Soft. Eng.*, May 2003.
17. Sun Microsystems. Enterprise JavaBeans specification. http://java.sun.com/ejb/.
18. Sun Microsystems. J2EE information and specification. http://java.sun.com/j2ee/.
19. The ObjectWeb Consortium. JOnAS: Java open application server.