Automating Service Dependency Management in a Service-Oriented Component Model

Humberto Cervantes and Richard S. Hall
Laboratoire LSR Imag, 220 rue de la Chimie
Domaine Universitaire, BP 53, 38041
Grenoble, Cedex 9 France
{Humberto.Cervantes,Richard.Hall}@imag.fr

ABSTRACT

This paper describes a mechanism to automate service dependency management in a service-oriented component model. The impetus behind this mechanism is not merely to eliminate complex and error-prone code from component-based applications, but also to deal with the phenomena of application building blocks that exhibit dynamic availability*, i.e., they may appear or disappear at any time and this is not under the control of the application. This intense focus on dynamic availability of building blocks is the result of the belief that applications of the future will become* context aware *in order to deal with building block proliferation. Such applications will employ* context-aware architectures *that use context (e.g., location, environment, user task) as a filter for including/excluding building blocks in/from their compositions. In this vision, automatic handling of dynamically available building blocks and their impact on application composition is critical. The service dependency management mechanism described in this paper is a starting point for such research and is implemented on top of the Open Services Gateway Initiative (OSGi) framework. The concepts and solutions it provides are sufficiently general for application in other service-oriented component models.*

Keywords

Service-Oriented Programming, Components, OSGi

1. INTRODUCTION

The need for techniques to automate component composition and to reason about such compositions continues to grow as more and more complex systems are being built using component-oriented approaches. Nowhere is this need more important than in systems built from dynamically available components. In this usage, *dynamic availability* refers to a situation where application building blocks may appear or disappear at anytime. An important aspect of dynamic availability is that it is not under application control, which requires that applications be ready to respond at any time to building block arrival and/or departure.

Computing trends, such as web services and pervasive computing, are making dynamically available building blocks commonplace. Web services push application functionality into network-based services and as a result push the inherent unreliability of distributed systems into ordinary client-side applications. Pervasive computing strives to embed computing power into almost all imaginable devices, each of which is able to offer services via wireless networks and other protocols. In both of these cases, service failures may occur, for example, when a server crashes or when a user simply walks out of wireless network range. These types of occurrences require that applications using the failed services deal with their dynamic departure. Likewise, applications may have to deal with dynamic building block arrival when servers or network connections are restored or when completely new services are discovered.

These scenarios are relevant to modern-day computing systems, but they also foreshadow a future in which continuous network connectivity is common and building blocks proliferate beyond the ability of applications to integrate efficiently and meaningfully with them. To deal with this coming building block proliferation, we envision a future where applications leverage context awareness in the form of *context-aware architectures*, where context (e.g., location, environment, user task) is used as a filter to determine which building blocks are included/excluded in/from an application's architectural composition at any given time. In this scenario, dynamic building block availability is the underlying issue to be resolved. Building blocks appear/disappear to/from an application based on their relevance to the current context. In turn, the application's composition must automatically adapt to these changes. Changes in context, and thus building block availability, are not under control of the application.

This paper describes our initial steps for addressing these issues by combining component-oriented and service-oriented concepts. The concepts from both of these areas are relevant since component orientation focuses on application building block definition and service orientation focuses on service dynamics and substitutability. The prototype platform for our research is implemented using the *Open Services Gateway Initiative (OSGi)* [11] framework, but the ideas are general enough for use in other service platforms. This prototype serves to demonstrate the types of automated composition techniques that are possible today, as well as to provide a foundation for experimenting with new compositional reasoning techniques in a dynamic setting. For example, compositional reasoning techniques could be used to ensure cost or security constraints as services are added and/or removed to/from an application.

The next section of this paper provides an overview of service-oriented programming, followed in section 3 by a short description of the OSGi framework. Section 4 presents the *Service Binder*, which automates service dependency management. Section 5 provides an overview of related work, while sections 6 and 7 discuss future work and conclusions, respectively.

2. SERVICE-ORIENTED PROGRAMMING

Service-Oriented Programming (SOP) is a programming methodology that promotes the concept of modeling solutions in terms of provided services that can be used by arbitrary clients. In this methodology, a service is a contract of defined behavior. In service-oriented programming, a client is not tied to a particular service provider, instead, the service providers are interchangeable [4]. Service-oriented solutions follow a similar pattern that consists of service providers, service requesters, and a service registry where services are published at run time by service providers and discovered by service requesters (see Figure 1). Several existing technologies implement service-oriented solutions, examples are Sun Microsystems' Jini [3] and web services [10].

Service-oriented programming has uniques traits when compared to other programming approaches. One trait is that services are dynamic in nature, meaning that they can be registered/unregistered to/from the service registry at any moment and clients must be prepared to cope with this situation. Existing service-oriented programming frameworks provide some type of notification API so that service clients can receive events and act upon the departure or arrival of services. Another trait is that service dependencies are unreliable and ambiguous. A service requester must be prepared to cope with situations where no required services are found or, on the other hand, multiple matching services are found. A final trait is that service requesters do not directly instantiate service instances, as is the case in object-orientation, for example. As a result, service requesters do not know whether they are interacting with a common service instance or with different instances providing the same service.

Component orientation, which focuses on creating re-usable software building blocks, is complementary to service orientation. A typical view of components is that they implement one or more provided interfaces, where an interface is a contract of functional behavior. In this sense, interfaces provided by components are very similar to service interfaces. This makes components an ideal candidate for implementing services, where a service is equated with a provided interface. The result is the concept of a *service-oriented component model*, which refers to a component model that registers the provided interfaces of its components into a service registry. Examples of service-oriented component models include the OSGi and Avalon [1].

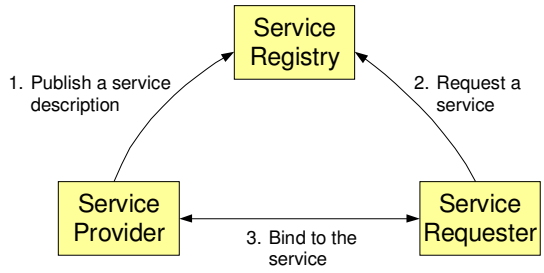

Figure 1: Pattern for service-oriented solutions.

3. THE OSGI SERVICES PLATFORM

The Open Services Gateway Initiative (OSGi) is an independent, non-profit corporation working to define and promote open specifications for the delivery of managed services to networks in homes, cars, and other types of restricted environments [11]. Specifically, the OSGi specification defines a service platform that includes a minimal component model, a small framework for managing the components, and a service registry. Services (i.e., Java interfaces) are packaged along with their implementations and their associated resources into *bundles*. Services are deployed, as bundles, into the OSGi framework via wide-area networks, i.e., the Internet.

The OSGi framework creates a host environment for managing bundles and the services they provide; a bundle is the physical unit of deployment in OSGi and is also a logical concept used by the framework to internally represent the service implementation. Concretely, a bundle is a Java JAR file that contains a manifest and some combination of Java class files, native code, and associated resources. The manifest of the bundle JAR file contains meta-data describing, among other things, the Java packages that the bundle requires or provides.

To use a bundle, it must be installed into the framework, which is handled automatically. An installed bundle is uniquely identifiable by either its bundle identifier (a number assigned dynamically by the framework when the bundle is installed) or by its location (which is an arbitrary character string used when installing the bundle). The location string is used to retrieve the bundle JAR file and is generally an URL. Since bundles are uniquely identified by their location string, it is not possible to install two or more bundles from the same location; thus, a bundle is essentially a singleton.

The management mechanisms provided by the framework allow for the installation, activation, deactivation, update, and removal of bundles. When a bundle is installed, it deploys a single component, called an activator, that can register and/or use services. When a bundle is activated, its corresponding activator component is instantiated by the framework. The activator implements activation and deactivation methods that are called to initialize and de-initialize it, respectively. In the activation/deactivation methods, the activator receives a context object, which gives it access to the framework and the service registry. The context allows the activator component to register services, look for other services, and register itself as a listener to different types of events that the framework may fire. When registering a service, the activator component may attach a set of attribute-value pairs to the service. Many different implementations of the same service may be registered by many different activator components and the associated service properties can be used to differentiate among them. To look for a service, an activator component uses the fully qualified service name and an optional filter in LDAP query syntax over the service properties. The instantiation of activator components is only performed by the OSGi framework; clients have no way to create component instances.

The state of a bundle can be changed at any moment while the framework is running. When a bundle is stopped, its associated activator component must unregister its services and release the services that it is using. Clients of the activator component's services must take care to observe the departure of the services. At that moment they must release the departing services and take any necessary corrective actions. Apart from these notifications, the OSGi framework does not provide any kind of support for service dependency management.

Two classes of service dependencies exist in OSGi: component-to-service and service-to-service. When a component depends on a service without itself providing services, we refer to the dependency as a component-to-service dependency. When the com-

ponent provides services and requires other services to provide its own service, we refer to the dependency as a service-to-service dependency. Writing the code to manage component-to-service and service-to-service dependencies is complex and error-prone; managing service dependencies involves concurrency and synchronization issues as well as tedious code to monitor the arrival and departure of any used services. The next section describes a mechanism to automate the tasks associated with service registration and service dependency management.

4. THE SERVICE BINDER

The Service Binder adds automatic service dependency management to OSGi and simplifies the task of writing service management code, greatly reducing the complexity of developing service-based OSGi applications. The Service Binder's goal is to automate the management of components and their service dependencies.

4.1 Approach

In general, the OSGi component model deploys one component per bundle; this component implements an interface, called `BundleActivator`, that defines two methods to activate and deactivate the component. The Service Binder changes this approach by allowing the bundle to deploy any number of component instances. In this context, a component type is associated with a Java class contained inside the bundle JAR file and a component instance is an instance of that class.

The Service Binder provides a generic bundle activator from which the developer only needs to create an empty subclass and provide component instance meta-data. The meta-data is in the form of an XML file, called an instance descriptor (described in detail in the next subsection), and is used to request which component instances the generic activator should create. A big benefit of this approach is that the application code no longer needs to reference or use the OSGi API; application code is completely isolated from the underlying OSGi service framework in most cases.

The goal of the generic activator, and consequently the Service Binder, is to create and manage each instance described in the instance descriptor file. For each component instance in the instance descriptor, the generic activator creates an *instance manager*. The instance manager has four responsibilities:

- dynamically monitor the component instance's service dependencies,
- create/destroy the component instance when its service dependencies are satisfied/unsatisfied,
- bind/unbind required services to/from the component instance when it is created/destroyed, and
- register/unregister any services provided by the component instance after its required services are bound/unbound.

Following from this, a component instance is always in one of two possible states: *invalid* or *valid* (see Figure 2). The invalid state means that the instance does not exist because at least one of the its service dependencies is not satisfied. The valid state means that the instance exists and that all of its service dependencies are satisfied and any provided services are usable. A component instance may also be destroyed, at which point it no longer exists. The ultimate goal of each instance manager is to keep its associated component instance in a valid state, but this is not always possible given the dynamic nature of services. As such, each instance manager actually represents the intention of creating a component instance. Each instance manager tries to maintain this intention throughout its lifetime.

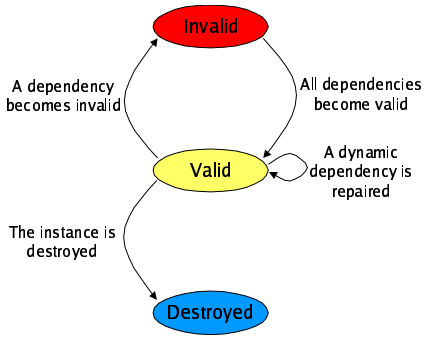

Figure 2: Instance life cycle.

4.2 Instance Descriptor

The instance descriptor is an XML file that contains meta-data describing the desired component instances to be created by the generic activator of the Service Binder (as described in the previous subsection). The instance descriptor file is contained in the bundle JAR file and extends the bundle's existing meta-data contained in the bundle's manifest file. Each component instance description includes the name of the class (contained in the bundle JAR file) that implements the component, the set of services implemented by the component, the set of properties associated with the services, and a set of service dependencies for the component instance.

Service dependencies are the most complex and, at a minimum, are characterized by the fully qualified service interface name of the required service. Two important characteristics define the precise behavior of a service dependency: *cardinality* and *binding policy*. Cardinality is useful for expressing optionality, such as a zero-to-one dependency, and also for expressing aggregation, such as a one-to-many dependency. Binding policy is as either *static* or *dynamic* and determines how run-time service changes are handled and how the component instance life cycle is managed. A static binding policy indicates that dependency bindings cannot change at run time, whereas a dynamic binding policy indicates that dependency bindings can change at run time. A static dependency is simpler to program than a dynamic one. In the static case, the required service is guaranteed to be present the entire time the instance is valid, while this condition is not guaranteed for dynamic dependencies.

```
<bundle>
  <instance class="org.foo.impl.SpellCheckServiceImpl">
    <service interface="org.foo.service.SpellCheckService"/>
    <property name="version" value="1.0" type="string"/>
    <requires
      service="org.foo.service.DictionaryService"
      filter="(Language=*)"
      cardinality="1..n"
      policy="dynamic"
      bind-method="addDictionary"
      unbind-method="removeDictionary"
    />
  </instance>
</bundle>
```

Figure 3: Instance descriptor.

As an example, if a service dependency is defined as zero-to-

many dynamic, then all available candidate services will be bound to the component when it is created and as those services arrive or depart at run time, they will be bound and unbound from the component as needed, respectively. Using this same example, if the dependency is changed to static, then arrivals of new services will be ignored and departures of bound services will result in the component instance first being destroyed and then re-created, if possible. At re-creation, the component instance may then be bound to any new services that had previously arrived. Thus, with a static binding policy, the component instance's life cycle is tied to its dependencies as well as changes to those dependencies, whereas with a dynamic binding policy the life cycle is only tied to its dependencies.

Figure 3 depicts a simple example of the XML-based instance descriptor. The different tags used in the instance descriptor are:

`<bundle>`

Tag that delimits the set of component instances contained inside the bundle. Several instances with service-to-service or component-to-service dependencies can be declared for one bundle. Simple service registrations are also supported.

`<instance>`

Defines the class of the component instance that will be created; the created instance will be used for binding/unbinding services and may also implement service interfaces itself.

`<provides> [optional]`

One of these tags must be included for each service interface that the component instance implements. If at least one these tags exist, it will result in a service registration for the defined service interfaces.

`<property> [optional]`

One of these tags must be included for each property that should be attached to the component instance. If a `<provides>` tag is present, the properties will also be attached to the service registration. The description of a property includes its name, value, and type (supported types are: string, boolean, byte, char, short, int, long, float, double).

`<requires> [optional]`

One of these tags must be included for each of the component instance's service dependencies. The properties of this tag are:

- service: the fully qualified interface name of the required service.
- filter: LDAP query to narrow search results.
- cardinality: 0..1, 0..n, 1..1, 1..n.
- policy: static or dynamic
- bind-method: The name of the method to call on the component instance to bind a service to it.
- unbind-method: The name of the method that to call on the component instance to unbind a service from it.

The above tags provide all of the meta-data needed to create and manage a component instance and its service dependencies. In addition to these tags, the Service Binder introduces a special tag for creating *component factory services*. A component factory service is used to create component instances at run time and is necessary because OSGi does not define its own component instantiation mechanism. The details of the component service factory tag are as follows:

`<factory> [optional]`

This tag contains a single `<instance>` tag (as described above). When seeing the factory tag, the Service Binder registers a special service, called a `FactoryService`. Clients may use this factory service to create multiple instances of the type associated with the factory. The Service Binder automatically manages any instances created from the factory service.

Despite the relative simplicity of these constructs, applications using the Service Binder exhibit interesting characteristics. For example, it is easy to describe a dynamic plugin-oriented system, such as a web browser, using a zero-to-many dynamic dependency between the browser and plugin services. This indicates that the web browser can work without any plugins and that it will automatically integrate or remove plugins as soon as they are installed or removed, respectively. Any application using the Service Binder can easily exhibit auto-adaptive behavior in response to dynamically installed and/or uninstalled components. Table 1 summarizes the semantics behind the different dependency definitions.

`1..1, static`	Instance is bound to one service, any change invalidates the instance
`1..1, dynamic`	Instance is bound to one service, changes do not invalidate the instance as long as it can be bound to another service
`1..n, static`	Instance is bound to at least one service, any change invalidates the instance
`1..n, dynamic`	Instance is bound to at least one service, changes do not invalidate the instance as long as the binding count is non-zero
`0..1, static`	Instance is bound to at most one service (i.e., optional), if it is bound, departure of the bound service invalidates the instance
`0..1, dynamic`	Instance is bound to at most one service (i.e., optional), the instance never becomes invalid
`0..n, static`	Instance is bound to all available services at the time of binding, any departure of a bound service invalidates the instance
`0..n, dynamic`	Instance is bound to all available services at the time of binding, as services arrive/depart they are bound/unbound to/from the instance, the instance never becomes invalid

Table 1: Different types of service dependencies.

The Service Binder is, itself, created and deployed as a bundle in the OSGi framework, which means that it can be used in any OSGi-compliant framework.

5. RELATED WORK

Related work includes on one side technologies related to service orientation and on the other side technologies related to dependency management. The former include service-oriented architectures such as Jini and Web Services, but also service-oriented component models such as Avalon. The latter include service composition mechanisms and dynamically reconfigurable systems.

Jini [3] is a set of specifications for a distributed service framework. The Jini infrastructure provides mechanisms for service registration, lookup, and notifications. Jini services can be added or removed from a registry, called the Jini federation, at any mo-

ment. Service clients, providers and the registry can reside in different machines, so all communication takes place through RMI. Jini supports distributed garbage collection through the concept of leasing, which grants a client to a service access to the service for a defined period of time. The Jini specification does not address service dependency automation.

Web services [10] are a service infrastructure that provides service description (WSDL) [15], discovery (UDDI) [2], and communication mechanisms (SOAP) [14] all based on XML, which makes them programming language independent. Web services technologies are not necessarily component oriented, but are complementary. Some recent work is looking into service dependencies, but not into service dependency management.

Avalon [1] is a service-oriented component model that is intended to be used as the infrastructure underlying server-side projects in the Apache organization. In Avalon there can be multiple service registries that allow groups of services to be created, however, service lookup mechanisms are very simple. In Avalon there is no support for service dependency management.

Service composition is another area that is related to this work. Combs [6] is studying service composition through a scalable, agent-based workflow language. Another system that supports service composition is eFlow [5]. eflow supports the specification, enactment, and management of composite e-services, modeled as processes that are executed by a service process engine. eFlows supports dynamic changes to the process schema and to the process state. Currently, all known service composition projects are taking a process or workflow approach instead of an architectural approach.

Dynamically reconfigurable systems generally employ an explicit architecture model and map changes on the model to the implementation. Examples of dynamic reconfigurable systems include ArchStudio [12] and [9]. Some dynamically reconfigurable systems are built by following new computing paradigms such as autonomic [8] or proactive computing [13]. Autonomic computing is oriented towards solving the problem of managing complexity by allowing systems to make decisions. To reach that goal, systems and their components must have the ability to self-monitor, self-heal, self-configure, and improve their performance. Proactive systems are based on prediction of user needs so that the system can take decisions with a minimum amount of user input. Both techniques are complementary and require some level of context awareness to take their decisions. Autonomic and proactive computing are relatively new fields that are still being defined and today only few applications built upon them are available. These applications, however, will require very dynamic and flexible infrastructures.

6. FUTURE WORK

The Service Binder currently uses a simple approach for resolving composition decisions, specifically it uses the fully qualified service interface name and an LDAP query over service properties to resolve composition dependencies. In the case where multiple choices for resolving a dependency exist, it simply selects the first one. The goal is to make this process more sophisticated.

One step is to use heuristics to further narrow the selection process when multiple choices exist. For example, when selecting a service to resolve a dependency, it might be more worthwhile to choose one whose implementing component has fewer dependencies of its own, because this might indicate that it will be easier to maintain. Along these lines, if two candidate implemention components have the same number of dependencies, it may be worthwhile to consider the types of dependencies (e.g., static or dynamic) and to choose the component with dynamic dependencies since this component may be more resilient to changes in its environment.

Standard heuristics, such as these, only go so far and cannot take advantage of higher level component knowledge. To make this possible, we plan to modify the Service Binder to allow pluggable “resolvers” that will be able to narrow selection choices using arbitrary reasoning techniques. This approach will provide an interesting testbed for new compositional reasoning approaches.

In addition to these issues, mechanisms are also necessary to deal with global composition issues. In general, the Service Binder deals with localized composition and even this simple approach makes it possible to create applications with very interesting characteristics, but it lacks some predictability as a result. For example, if two candidate services are available, the one chosen to resolve the dependency is non-deterministic. The Service Binder's focus on localized dynamic availability and substitutability must be mirrored at an architectural level with a service-oriented architecture. A service-oriented architecture must be flexible to support dynamic availability, but must also try to attain some level of predictability.

We envision a hierarchical service-oriented architecture model that allows compositions to be used as components in higher level compositions. In this vision, dynamic changes in component availability will percolate up and down the composition hierarchy to repair and adapt the application as components arrive and depart. This scenario implies the need for even more advanced compositional reasoning techniques to determine component “fit” as well as to verify global properties of the application.

7. CONCLUSION

Computing trends such as web services and pervasive computing are increasing the importance of application building blocks that exhibit dynamic availability due to their inherent unreliability that is not under application control. Further, these trends are leading to a proliferation of application building blocks that require new techniques for composing applications. Our view is that applications of the future will use context-aware architectures to automatically adapt their compositions according to the current usage context in order to make building block integration decisions. At the heart of this vision is the need to automate application composition.

This paper introduced a mechanism to automate service dependency management in a service-oriented component model. Dependencies between components and services are described declaratively in an XML file that is deployed along with the component. This file is parsed by a mechanism called the Service Binder, which manages the components and their dependencies automatically. The benefits of the approach are immediate: dependency management code is separated from the service implementation, the need to write complex and error prone dependency management code is eliminated, and the application is isolated from the underlying service framework.

The mechanism described in this paper was implemented on top of the OSGi service-oriented framework, but the characterization of service dependencies is also valid for other service platforms. The problems that OSGi programmers encounter are similar to those that a Jini or web services programmer must solve.

This work belongs to a bigger project, called Gravity, whose goal is to define a hierarchical service-oriented component model that fully supports dynamic availability of application building blocks. The Service Binder is the underlying mechanism that handles the dependencies among components and services. Even with the relatively simple concepts introduced by the Service Binder, resulting applications have interesting characteristics with respect to run-time adaptability. The Service Binder is available at: `http://gravity.sourceforge.net`.

8. REFERENCES

[1] Apache.org, "The Avalon Framework", http://jakarta.apache.org/avalon

[2] Ariba Corp., IBM Corp., and Microsoft Corp. "UDDI Technical White Paper," September 2000.

[3] K. Arnold, R. O'Sullivan, W. Scheifler, and A. Wollrath. "The Jini Specification," Addison-Wesley, Reading, Mass. 1999.

[4] G. Bieber, J. Carpenter, "Introduction to Service-Oriented Programming", September 2001, Online whitepaper at http://www.openwings.org/download.html

[5] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. "Adaptive and Dynamic Service Composition in eFlow," Technical Report HPL-2000-39, Hewlett-Packard, 2000.

[6] N. Combs. "Reliable Recruitment and Assembly of Peer-to-Peer Services and Distributed Workflow," Working Conference on Complex and Dynamic Systems Architecture, December 2001.

[7] C.R. Hofmeister. "Dynamic Reconfiguration of Distributed Applications," Ph.D. Thesis, Computer Science Department, University of Maryland, College Park, 1993.

[8] P. Horn. "Autonomic Computing," IBM Manifesto, October 2001.

[9] F. Kon. "Automatic Configuration of Component-Based Distributed Systems," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

[10] H. Kreger. "Web Services Conceptual Architecture (WSCA 1.0)," IBM Software Group, 2001.

[11] Open Services Gateway Initiative. "OSGI Service Platform," Specification Release 2.0, October 2001.

[12] P. Oreizy and R.N. Taylor. "On the Role of Software Architectures in Runtime System Reconfiguration," IEEE Proceedings-Software, vol 145, no. 5, October 1998.

[13] D.L. Tennenhouse. "Proactive Computing," Communications of the ACM, Vol. 43 No. 5, pp.43-50, May 2000.

[14] World Wide Web Consortium. "Simple Object Access Protocol (SOAP) 1.1," W3C Note 08, May 2000.

[15] World Wide Web Consortium. "Web Services Description Language (WSDL) 1.1," W3C Note 15 March 2001."