Abstract

This paper describes a framework that supports the construction of component-based applications that are able to adapt to the dynamic availability of their constituent components. These applications are built using a service-oriented component model, which is a component model that includes some concepts from service orientation. The components defined by this model define information that allows bindings among components to be created and managed externally. This paper focuses on how auto-adaptive applications are built and how they adapt to dynamic changes in component availability. It also describes an example scenario and two real-world examples where this framework has been used. The framework is built on top of the OSGi service-platform.

Keywords

Component-Orientation, Service-Orientation, Dynamic Availability, Auto-Adaptation, OSGi

1 Introduction

This paper describes a framework that supports the construction of component-based applications that are able to adapt to dynamic changes in the availability of their constituent components. Dynamic availability of constituent components is the situation where components become available or unavailable during application execution. This concept supposes the existence of an actor that introduces or removes components from the environment in which the application is executing and this actor is not under the control of the application.

The motivation for this research is to simplify the creation of applications that are capable of autonomously reacting to changes in component availability, such as a web browser that is capable of automatically incorporating plugins to visualize particular content, and also to propose mechanisms to support novel computing approaches such as context-aware computing [Dey2000]. In context-aware computing, contextual information, such as the user’s location, is used to direct the functionality and/or behavior of an application. For example, if a user enters an airport, an application inside his PDA may incorporate services offered by a component obtained from the airport's wireless network, such as one that provides flight information. When the user leaves the airport, these components are no longer available.

The basic adaptation mechanisms provided by this framework were presented in [Cervantes2003]; that paper focused on service dependency management concepts and did not discuss how applications are built or adapted. This paper focuses on auto-adaptive applications; its outline is the following: section 2 presents the concepts of the service-oriented component model, section 3 presents an example scenario, section 4 briefly discusses how the service-oriented component model is built on top of the OSGi service platform. Section 5 presents two industrial applications where the framework discussed here has been used. Finally, section 6 presents related work followed by future work in section 7 and a conclusion.

2 Service-oriented component model

This section describes the fundamental characteristics of a service-oriented component model.

2.1 Foundations

The two approaches upon which the service-oriented component model is based are component and service orientation.

2.1.1 Component orientation

Although there is no universal agreement on the definition for the term component, the definition formulated by Szyperski [Szyperski1998] is widely referenced in literature:

“A software component is a binary unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

In the context of this paper (and following the majority of industrial component models, such as EJB [Sun2001], CCM [OMG1999] and COM [Box1998]), components are seen as entities similar to classes in the sense that they can be instantiated and that their instances can be stateful. A composition is created by first creating some component instances, and then by customizing and connecting these instances to each other in some appropriate fashion. To build
an application, an assembler (the third party in the definition above) defines a description that directs the creation of a composition during execution.

To support composition by third parties along with binary delivery, a component is generally accompanied by information that describes the external structure of its instances, such as provided and required functional interfaces and modifiable properties. The independent deployment aspect of components refers to the fact that a component is delivered in a package that contains binary code along with resources required by the component, which include artifacts such as images and libraries.

Finally, components require an execution environment that provides run-time support to the component instances, normally through what is known as a container. Run-time support includes life-cycle management of component instances and support for non-functional characteristics such as transparent remote communication, security, persistence, and transactions.

2.1.2 Service orientation

Service orientation focuses on “how services are described and organized in a way that supports the dynamic discovery of appropriate services at run time.” [Burbeck 2000]. A service is defined in [Bieber2001] as a contractually defined behavior, which in practice is realized by a service interface implemented by a service object. To support dynamic discovery, service orientation follows an architectural style where three different actors interact; these actors are service providers, service requesters, and one or more service registries. Service providers publish a description of the services they provide into the service registry so that clients can locate them. Service requesters interrogate the registry to find service providers based on a service description. If suitable service providers are present in the service registry at the moment a request is made, the registry returns a reference to one or more service providers so that the service requester can select and bind itself to any of them. During binding the service provider returns a reference either to itself or to a service object that implements the service interface.

Service orientation promotes the idea that a service requester is not tied to a particular service provider; instead, providers of the same service are substitutable. In service orientation, services providers and requesters are typically independent entities; as a consequence, and from the requester's point of view, services exhibit dynamic availability as they are added to or removed from the service registry.

A service-oriented application is built by composing service objects based on their service interfaces. A service composition is an abstract description that becomes concrete at execution time, when service providers are located and integrated into the composition. A service-oriented application needs to cope with the potential arrival or departure of services it uses as it executes. To support this task, some service platforms provide notification mechanisms that announce changes to services that are present in the registry.

2.2 Service components

The main concept in a service-oriented component model is that of a service component, whose elements are depicted in figure 1.

![Figure 1: Elements of a service component.](image)

2.2.1 Provided service interfaces and service properties

A service component declares a variable number of provided service interfaces, which represent services provided by the component's instances during execution. Services provided by component instances are published, as a service interface together with service properties, in a service registry and service requesters bind directly to the instances when the services are requested.

A particular component instance may be registered multiple times in the service registry if the component provides multiple service interfaces. When a service component does not provide any service interfaces, the instance is not registered and other instances cannot interact with it through services; this instance can however use services provided by other instances.

2.2.2 Component implementation and required service interfaces.

A component implementation contains code that implements the provided service interfaces. Furthermore, a component implementation can compose services, that is, it can use services provided by other component instances. When this occurs, an instance of the component is bound, during execution, to other component instances that provide the services to be composed. In this case, the service component also needs to declare that it requires a set of service interfaces corresponding to the services that are composed inside the implementation. One or more component implementations make up an application.

2.2.3 Control interfaces and implementation dependencies

Control interfaces are required to support instance life-cycle management and to allow instances to participate in dynamic reconfiguration operations; the dynamic reconfiguration operations are described in section 3. Implementation dependencies represent dependencies from a service component towards artifacts such as resources or libraries. Dur-
ing execution, these two elements are only visible to the execution environment.

2.3 Automated service dependency management

Aside from the introduction of service concepts into a component model, the second pillar of the approach presented in this paper is the concept of automated service dependency management. Service dependency management is the activity of creating and changing the bindings among a set of component instances that form an application. Automation of this activity is required as component instances are introduced and removed from the environment where an application is executing.

As component instances are added or removed, service compositions need to release or to incorporate services provided by these instances. Automated service dependency management finds substitutes for services from departing instances or incorporates services provided by new instances into the running application.

2.3.1 Dependency properties

To support automated service dependency management, required service interfaces are characterized by a set of dependency properties that include the following informations:

- **Cardinality.** Expresses optionality, indicating that a binding is optional or mandatory, and multiplicity, indicating that the required service interfaces represents a single or aggregate binding.
- **Policy.** Expresses whether change in a binding invalidates an instance or not. Policy is either static, indicating that all changes to the binding invalidate the instance, or dynamic, indicating that changes to the binding are allowed. Invalidation of a component instance is discussed in the next subsection.
- **Filter.** Expresses a constraint on the creation of a binding to a subset of available providers using a query over service properties. This allows unpredictability to be reduced; unpredictability is discussed in section 3.2.2.

2.3.2 Instance manager

During execution, each component instance is managed by an instance manager, and the creation of a component instance and of its manager are simultaneous. The creation of a component instance can, however, is considered “intentional”, since the instance that is created is initially invalid and will not actually exist until its dependencies are satisfied. An invalid instance does not provide its services and is not executing, while a valid instance provides its services and is executing.

For an instance to become valid, the services that it requires need to be bound to it. Service binding, which includes binding creation and management, is the responsibility of the instance manager who also controls the instance’s life cycle, manages service registrations, and maintains the instance’s validity. Bindings between component instances are created following a service-orientation interaction style, where the instance manager queries a service registry to find services provided by other instances. Binding management occurs when the instance manager receives notifications announcing changes in the service registry. Figure 2 illustrates these concepts; this figure also shows that the instance manager has access to an internal view of a component instance, which contains control interfaces and implementation dependencies, while instances in the composition are seen only from their external view, which includes provided and required service interfaces and service properties.

Figure 2: A component instance and its manager

The instance's life cycle is managed according to the states depicted in figure 3. The states and the transitions that are part of the instance manager's responsibilities are depicted in light gray and as solid arrows, respectively. Immediately after creation, the instance manager enters the initialization step, where it binds the instance it manages to other instances that provide required services. If bindings that are mandatory cannot be created, initialization fails and the instance manager awaits for notifications about the arrival of adequate services and then tries to repeat the procedure again.

When initialization finishes, the instance is activated (notified by a call to a method in a control interface) and its provided service interfaces are registered in the service registry; at this point the instance is valid. Once an instance is valid, the instance manager enters the execution step. At that point, the instance manager receives notifications that may trigger a reconfiguration, i.e., changes in the instance's bindings. If reconfiguration succeeds, the execution continues until a new reconfiguration occurs. If reconfiguration fails, the instance is invalidated.

During invalidation, the instance's services are removed.
from the service registry and the instance is notified that it is being invalidated. After this, the instance becomes invalid and the manager enters a new management cycle where it tries to initialize the instance anew. This cycle represents the fact that the instance manager's intent is to maintain the validity of the instance it manages continuously. The cycle ends when an instance is destroyed; this is represented by the dashed arrows. Before being destroyed, and if the instance is valid, it is subsequently invalidated.

The success of the initialization and reconfiguration activities depends on the dependency properties. Initialization always succeeds when only optional or non-mandatory required service interfaces are present in a component instance, otherwise bindings need to be created for mandatory required service interfaces. In order for reconfiguration to succeed without invalidating the component instance, the required service interface's dependency policy must be dynamic, otherwise any change will invalidate the instance. Even if the dependency policy is dynamic, reconfiguration will need to invalidate a component instance if the dynamic changes violate a mandatory cardinality value for a required service interface.

3 Assembly and adaptation of an application

During execution, an application is constructed as a set of interconnected component instances. As the application executes, changes in dynamic availability results when new component instances are introduced into the system or existing instances are destroyed. The dependency properties associated with the component's required service interfaces and the individual management of component instances result in applications that are auto-assembled and that adapt automatically to component instance availability during execution. The changes that such an application can handle include the addition, removal, and substitution of a component instance.

3.1 Execution scenario

This section presents a scenario to further illustrate the types of adaptations that can occur in an application due to dynamic changes in the availability of its constituent components.

3.1.1 Initial assembly

An application is typically built out of a “core” component instance that contains the main service composition that guides the application's execution. Other component instances provide the services used by the core component and these instances can themselves require services provided by other instances. The assembly process of such an application begins as instances are created inside the execution environment of the service-oriented component model. The execution of the application starts the moment the main instance becomes valid.

According to the life-cycle management characteristics previously described, instance validation occurs in a specific order. Supposing that at the time the application instances are created, the system contains no other instances that provide services required by the application, the order of validation is the following:

1. Instances with optional or no required service instances are validated first.
2. Instances with mandatory required service instances are then validated if the services they require have become available at that point.

Figure 4.a depicts a word-processor application example that is built out of a core component instance (main from WordProcessorComponent) that uses services for spell checking and printing purposes. The core component instance, depicted at the left from its external view, has two required service interfaces: the first one is the spell checking service and is characterized as being mandatory, single, and dynamic; the second one is the printing service and is characterized as being optional, single, and dynamic. In the figure, the core instance is bound to two instances that provide the required spell checking and printing services, checker (from SpellCheckComponent) and printer (from PrintComponent), respectively. The checker instance itself requires a service interface for a dictionary service. Its required service interface is characterized as mandatory, aggregate, and dynamic. The checker instance is bound to another component instance that provides a dictionary service (frenchdict from FrenchDictComponent).

If these four instances were created simultaneously, frenchdict and printer would be validated initially (in no particular order), after the frenchdict became valid, the checker would be configured and validated. As a result of this validation, main would be configured and validated. At this point the application would begin execution.

3.1.2 Instance addition

Figure 4.b shows the word-processor application after a new instance providing a dictionary service (englishdict from EnglishDictComponent) became available. The characteristics of the required service interface of the spell check component (aggregate and dynamic) allow new bindings to be created and, as a result, the englishdict instance is added to the application.

3.1.3 Instance removal

Figure 4.c shows the word processing application after the removal of the printer instance. The destruction of the binding does not cause checker to become invalid since the required service interface is characterized as optional and dynamic.

3.1.4 Instance substitution

Instance substitution may result as a consequence of an instance removal that satisfies a required service interface that is characterized as mandatory, single, and dynamic. In the word processor example, this is the case for the SpellCheckService of the main instance. The service reg-
Figure 4: Word processing application evolution.

Service registry

- DictionaryService
- PrintingService
- SpellCheckService

Figure 4.a contains an entry for an additional SpellCheckService than the one being used by the application; since the required service interface is a single binding, only a single connection is made to one of the available spell checker services. In figure 4.d, however, the checker instance is removed. In response, the instance manager for the main instance looks for a substitute, which is fulfilled by the other available component instance. The instance manager creates a binding to the new spell checker (checker2 from EnglishSpellCheckComponent) and the reconfiguration succeeds. Notice that this checker2 is itself bound to one of the previously used dictionaries.

3.2 Discussion

This approach for auto-adaptable applications has consequences for maintaining an entire application's validity as well as some limitations; some of these issues are discussed briefly in the following subsections.

3.2.1 Application destruction and repair

A situation that can occur as a result of an instance being destroyed is the triggering of a “chain reaction” that leads to the invalidation of the main component and, thus, the entire application. An example of this situation in the word processor example is if the system is in the state depicted in figure 4.a and then suddenly the frenchdict instance is destroyed. At that moment the instance manager for the spell checker looks for a replacement for the dictionary service, but, since none are available, the checker instance would become invalid. This invalidation would then invalidate the main instance, since no other spell checker is available. This situation would cause the different instance managers to enter new management cycles. As a consequence, as soon as a new instance providing a dictionary service becomes available, the spell checker will be re-activated, followed by the main instance, and then the application as a whole.

3.2.2 Limitations of this approach

Two limitations of this approach are:

- **Unpredictability.** Certain situations lead to unpredictability with respect to the creation of bindings among component instances. This situation occurs, for example, when a required service interface is categorized as being single, but at the time the instance manager creates the binding, multiple candidates are present. Unpredictability can be reduced by using filters in required service interfaces, but this is not sufficient.

- **Circular dependencies.** Given that instance managers only have localized information, a deadlock situation occurs if an instance has a mandatory requirement on a service provided by another instance that itself requires a service provided by the first instance. In this case, neither of the instances can be validated as both instance managers are waiting for the other instance's service to become available. Simplistically, this situation can be overcome by defining one of the required service inter-
faces as being optional and dynamic, but a more general solution requires cycle detection during validation.

4 Implementation

The execution environment of the service oriented component model has been implemented on top of the OSGi service platform. The implementation of the automated service dependency management mechanism is called the Service Binder [Cervantes2003].

OSGi [OSGi2001] is a centralized service platform that provides a service registry along with a notification mechanisms necessary for the component model's execution environment. OSGi also provides deployment mechanisms that allow deployment units, called bundles, to be continually installed, removed and updated inside the platform. Once a bundle is installed in the platform, it can be started if certain code dependencies that are associated to the bundle are validated.

The benefits of using the Service Binder to develop OSGi applications include the introduction of the service component concept along with the extraction of service dependency management and adaptation logic from the component code.

4.1 Deployment

OSGi's bundles are used as deployment units both for service components and for the Service Binder. A bundle can contain one or more service components that are declared in a component descriptor. The bundle contains the component implementations along with resources such as binary files or libraries.

```xml
<component class="org.gravity.webbrowser.WebBrowserImpl">
  <provides service="org.gravity.services.WebBrowser" />
  <property name="version" value="1.0" type="string"/>
  <requires
      service="org.gravity.services.WindowManager"
      filter="cardinality="1..1"" policy="static"
      bind-method="setWindowManager"
      unbind-method="unsetWindowManager" />
</component>
```

Figure 5: A service component description.

4.2 Component descriptor

A component descriptor is an XML file that contains a series of service components descriptions that correspond to the service components that are deployed inside a bundle. A service component description is depicted in figure 5. Inside the component tag of the description, the class attribute refers to the component implementation, which is a Java class. The service attribute inside the provides and requires tags corresponds to service interfaces that are implemented as Java interfaces. The requires tag contains the different dependency properties:

- the filter, which is written in an LDAP syntax,
- the cardinality, which represents both optionalty, at the lower end as a 0 or a 1, and aggregation at the upper end, as a 1 or a n (e.g., a mandatory, aggregate required service interface is declared as 1..n),

- the policy, which can be static or dynamic, and
- the binding methods, which are methods used by the instance manager to bind or unbind its instance.

4.3 Instance creation and destruction

The construction of applications on the OSGi platform is realized through the concept of deployment instances, which represents a component instance that is created at the moment a bundle is activated and that is destroyed when the bundle is deactivated. An application is assembled by launching an OSGi platform with a particular set of installed bundles or by installing bundles as the system is running. The application evolves as deployment activities, realized through the platform's deployment mechanisms, occur during execution.

5 Evaluations

This section describes two industrial scenarios in which the techniques for auto-assembly and auto-adaptation described in this paper were used: the Service Binder implementation was used as the foundation for these applications.

5.1 Device monitoring at Schneider Electric

The Service Binder was used at Schneider Electric for a research project oriented around electric device monitoring. In this project, a series of electric devices are connected to a bus that is itself accessible to a gateway running an OSGi platform. A series of monitoring components inside the system are in charge of polling devices and producing notifications when exceptional situations occur. Requirements for this system are that it must run continuously and that new monitoring components need to be added to or removed from the running system as new devices are connected or disconnected to or from the bus.

Figure 6 represents the architecture of the system. The different component instances that form its architecture are:

- **Business Objects**: These components contain monitoring logic. Business objects provide a service (BOService) to allow their activation and may require services to produce notifications, such as creating a log or sending an e-mail.
- **Scheduler**: This component coordinates the application. Its goal is to periodically activate the different business objects that are available in the system. The presence of the scheduler is necessary due to material limitations, which restrain the business objects to run simultaneously.
- **Poll**: The poll component provides a service to the business objects so that they can obtain data from the different devices attached to the bus.
- **ModBus**: The bus connects to the physical bus where

---

1 http://www.schneiderelectric.com
devices are connected.

- **Mailer and Logger**: These components provide ways to produce notifications via e-mail and a log, respectively.

Support for the runtime addition or removal of business objects is achieved through the required service interface in the scheduler towards the BOService, which is characterized as optional, aggregate, and dynamic, and by the required service interface in the business objects towards the poll. In this application, business objects and notification mechanisms are introduced into the system as a result of continuous deployment activities.

### 5.2 VersaTest client

Another application of the Service Binder was created at a company called Ascert, which manufactures an application oriented towards testing OLTP systems, called VersaTest. VersaTest runs as a server and a client for the VersaTest system. The VersaTest client application is the support for multiple configurations, where a configuration consists of the core and a particular set of tools.

To support multiple configurations, the VersaTest client launches the OSGi platform with varying sets of installed bundles. When the platform is launched, the particular configuration is assembled dynamically. The VersaTest client also supports the addition and removal of tools during execution and also updates as the tool is running, but these capabilities are not currently used in the commercial version of the application.

### 6 Related work

Related work includes component models and service platforms, along with techniques to compose services and create auto-adaptive applications.

The OSGi's wire admin service provides a mechanism to compose OSGi services. Services are categorized either as producers or consumers and are connected to each other through connectors called wires. A wire is defined for a specific producer and consumer pair and the wire is realized at the moment both services become available.

The work of [Kon2000] describes an approach to support the automatic configuration of distributed component-based systems. In this work, an entity similar to the instance manager, called a component configurator, contains information regarding the connections from one component to other components. Reconfiguration is directed by an agent that interacts with the different configurators.

The work of [Georgiadis2002] presents a framework to create self-organizing distributed systems. In this approach, component instances are managed independently, and their connections are modified when component instances are introduced or removed to or from the system according to constraints defined as an architectural style written in an ADL. This approach, does not address the problem of unpredictability.

### 7 Future work

Ongoing work includes studying mechanisms to create applications where multiple instances from the same component are used. This situation is not trivial since it exacerbates the unpredictability problem because all instances of
from same component look identical as far as instance managers are concerned; thus, knowing which instance to choose among multiple instances of the same component is complex. The current approach being investigated to deal with this issue is by placing instances into resolution scopes that constrain the creation of connections to only those candidates in the same scope. This work, along with a mechanism to support management at the instance composition level are discussed in [Cervantes2004].

Other work includes an interactive development environment where component instances are introduced in a design canvas through drag-and-drop gestures. This environment differs from other component assembly environments in that the assembler does not “draw” connections between instances. Although a prototype for this environment has been built, it is at a proof-of-concept stage and improvements to it include the possibility of automatic creation of scopes and also techniques to suggest to the developer which components can be dropped into the design area so that instances are successfully validated.

8 Conclusions

This paper presented a framework to create component-based applications that are assembled and adapted dynamically as component instances are introduced or removed from the application execution environment. In this framework, each component instance is managed independently and its bindings are adapted at run time based on information associated with its required service interfaces. Instance managers strive to maintain an instance's validity with respect to its bindings.

Applications built from this framework are capable of adding new component instances or releasing/substituting departing component instances. They also exhibit self-repairing characteristics since instance managers create and repair component instances and consequently the application as a whole as components become available or unavailable at run time. This framework was implemented on top of the OSGi framework and was successfully used in two different industrial applications. Initial feedback from users is very encouraging.

9 References