A Framework for Constructing Adaptive Component-Based Applications: Concepts and Experiences

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 the experience of building component-oriented applications with a framework that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. The framework's approach is to define a service-oriented component model, which is a component model that includes concepts from service orientation and an execution environment that provides automatic adaptation mechanisms. This paper focuses on an example scenario and two real-world examples where this framework has been used.

1. Introduction

This paper describes the experience of building component-oriented applications with a framework, called Gravity [4], that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. Dynamic availability is a situation where functionality provided by components may appear or disappear at any time during application execution and this outside of application control.

In this paper, dynamic availability is conceptually caused by continuous deployment activities that introduce and remove component functionality into the execution environment. Continuous deployment activities represent the installation, update, and removal of components during execution by an actor that may or may not be the application itself. The constituent components of an application are directly impacted by these activities, since they are likely to have dependencies on functionality provided by other components in order to provide their own functionalities. If some required functionality is removed, a component may not be able to provide its own functionality any longer; on the contrary, newly arriving functionality may enable another component to provide its own functionality.

The traditional component-oriented approach is based on the ideas that an application is assembled from reusable building blocks [9] (i.e., components) available at the time of assembly and that during execution, components are not spontaneously added or removed. As a consequence, dynamic availability is not supported explicitly in component models. Of course, it is possible to support dynamically available components through programmatic means, but this results in mixing both application and adaptation logic into the application code. In order to simplify this situation, component orientation should incorporate concepts from other approaches that explicitly support dynamic

availability of functionality, such as service orientation. In service-oriented computing, services provide functionality that can be published and removed from service registries [4] at any time.

This paper discusses the experiences of building applications using the Gravity framework that provides a service-oriented component model and an associated execution environment. In this approach, functionality provided by components and their subsequent compositions are realized using service-orientation concepts. Special attention is paid to one aspect of the Gravity framework, called the Service Binder, that manages adaptation logic at run time. Applications built using this mechanism are capable of assembling and adapting autonomously. The basic approach of the Service Binder was presented in [3]; this paper describes how applications using this mechanism are built and how they adapt.

The remainder of the paper is structured as follows: section 2 discusses management of dynamic availability, section 3 presents an example scenario, section 4 presents two application scenarios where these concepts were applied, and section 5 presents related work followed by section 6 with conclusions and future work.

2. Managing dynamic availability

In a service-oriented component model, components provide and require functionality, where the functionality is modeled as a service that is published and discovered at run time using a service registry inside the execution environment. Gravity's execution environment also manages dynamic availability by creating and destroying bindings among component instances using the service-oriented interaction pattern of publish-find-bind [3]. To enable this, a component provides information (*dependency properties*) about its service dependencies to the execution environment for run-time management.

In this approach, a component instance is either *invalid* or *valid*. An invalid instance's service dependencies are not satisfied and it is unable to execute and to provide its services, while a valid instance's service dependencies are satisfied and it is able to execute and provide its services. All instances initially start in an invalid state and become valid if their service dependencies are satisfied. At run time, instances may alternate between valid and invalid as required functionality is added to and removed from the execution environment, respectively. As such, the creation of a component instance is intentional, since the execution environment of the framework constantly tries to maintain the validity of the instance until it is explicitly destroyed.

To facilitate management, a component is described with a component descriptor; an example descriptor is depicted in figure 1. Inside the `component` tag of the descriptor, the `class` attribute refers to the component implementation, which is a Java class. The `service` attribute inside the `provides` and `requires` tags corresponds to provided and required services, respectively, which are described syntactically as Java interfaces. Additionally, the `requires` tag declares additional information including:

- **Cardinality.** Expresses both optionality and aggregation. In a component descriptor, the lower end of the cardinality value represents optionality, where a '`0`' means that dependency is optional and '`1`' means that it is mandatory. The upper end of the cardinality value represents aggregation, where a '`1`' means the dependency is singular and '`n`' means that it is aggregate.
- **Policy.** Determines how dynamic service changes are handled: a *static* policy indi-

```
<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.Plugin"
      filter=""
      cardinality="0..n"
      policy="dynamic"
      bind-method="addPlugin"
      unbind-method="removePlugin"
    />
</component>
```

Fig. 1. A service component description

cates that bindings cannot change at run time without causing the instance to become invalid, whereas a *dynamic* policy indicates that bindings can change at run time as long as bindings for mandatory dependencies are satisfied.

- **Filter.** Constrains candidate providers of a required service to those matching a query (written in LDAP query syntax); the query is issued over the service properties associated with components using the `property` tag.
- **Bind/unbind methods.** These methods allow the execution environment to set/unset references to the required service, i.e., create bindings.

During execution components may be installed or removed and, as a result, component instances are automatically created and destroyed by the execution environment. When an instance is created, it is associated with an *instance manager* that acts as a container responsible for managing instance's life cycle. The instance manager uses the execution environment's service registry to resolve and monitor its instance's service dependencies. When all of the instance's service dependencies are satisfied, the instance manager activates the instance and publishes its services; at this point the instance is valid. During execution, the service registry notifies instance managers about changes in service availability. An instance manager may need to reconfigure its bindings according to dependency properties in response to service registry notifications. If reconfiguration is not possible, the instance manager invalidates its instance and continuously tries to make it valid again.

3. Assembly and adaptation of an application

In a service-oriented component model, an application is constructed as a set of interconnected valid component instances at execution time. Such an application assembles and adapts autonomously as its constituent component instances are connected by their respective instance managers. Autonomous management for dynamic availability is discussed below, along with the main limitation of this approach.

3.1. Autonomous management for dynamic availability

Initial assembly. An application in this framework is typically built out of a “core” component instance that guides the application's execution. Other component instances

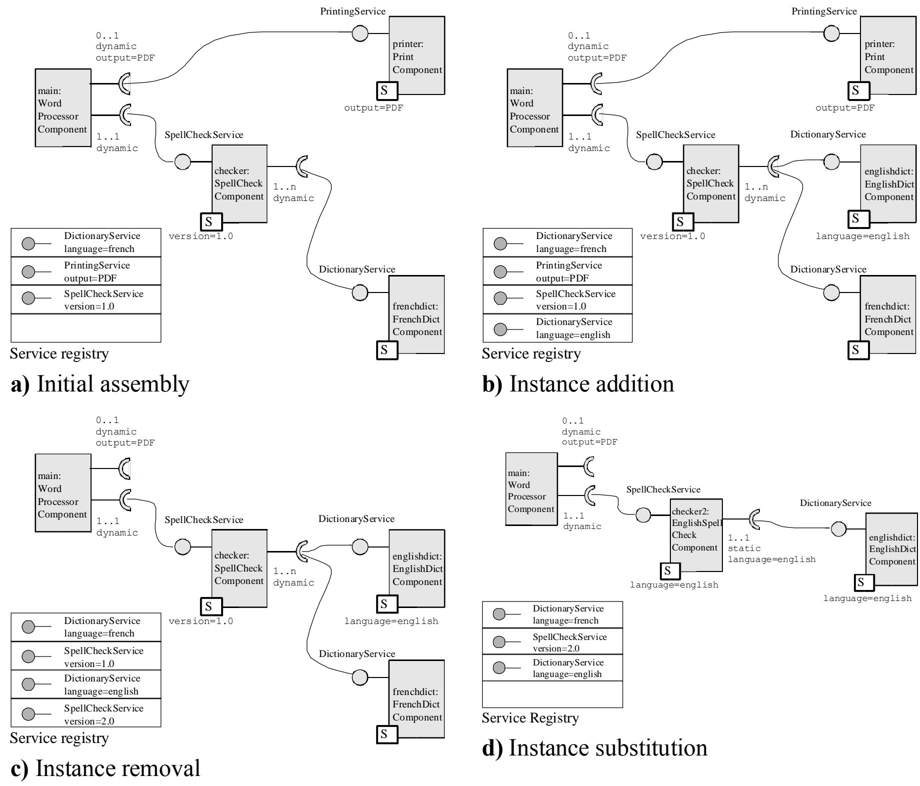


Fig. 2. Word processing application evolution

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 core instance becomes valid. The validation of the core instance occurs in a specific order that obeys the service dependency characteristics of the individual components. Specifically, the order is:

1. Instances with optional or no service dependencies are validated first.
2. Instances with mandatory service dependencies are validated if the services they require become available due to newly validated services.
3. The second step repeats as long as new services are introduced that resolve additional mandatory service dependencies.

Figure 2a 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 *main* component instance, depicted at the left of the figure, has two service dependencies; the first dependency is on a spell checking service and is mandatory, singular, and dynamic, the second dependency is on a printing service and is optional, singular, and dynamic. In the figure, the *main* 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 dictionary service; this dependency is 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 are created simultaneously, *frenchdict* and *printer* are validated first in no particular order, after which the *frenchdict* becomes valid, then the *checker* becomes valid, and finally *main* becomes valid. At this point the application begins execution.

Instance addition. Figure 2b shows the word-processor application after a new instance providing a dictionary service (*englishdict* from `EnglishDictComponent`) becomes available. The characteristics of the spell checker component's service dependency (aggregate and dynamic) allow new bindings to be created and, as a result, the *englishdict* instance is added to the application.

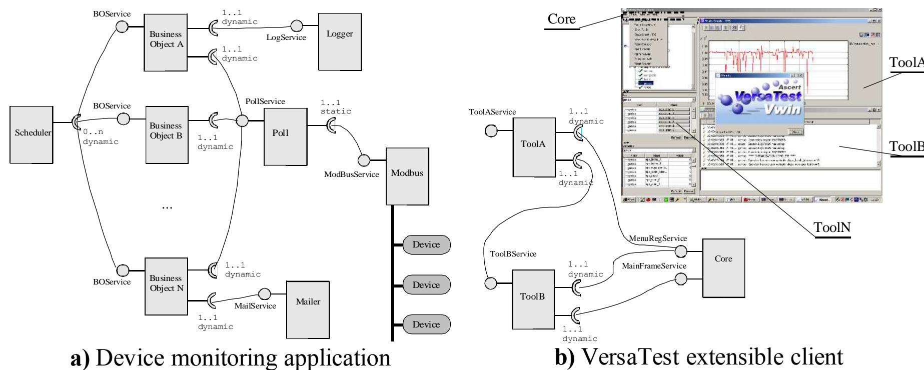
Instance removal. Figure 2c shows the word-processor application after the removal of the *printer* instance. The destruction of the associated binding does not cause *main* to become invalid since the required service interface is characterized as optional and dynamic.

Instance substitution. Instance substitution results from the removal of an instance that satisfies a service dependency that is mandatory, singular, and dynamic. In the word-processor application, this is the case for the `SpellCheckService` of the main instance. The service registry in figure 2c contains an entry for an additional `SpellCheckService` than the one being used by the application; since the service dependency is singular, only a single binding is created to one of the available spell checker services. In figure 2d, 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 *checker2* is already bound to the existing English dictionary.

Application invalidation 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 core component and, thus, the entire application. This situation occurs if the word-processor application is in the state depicted in figure 2a and the *frenchdict* instance is destroyed. The instance manager for the spell checker will look for a replacement, but the *checker* instance becomes invalid since no alternatives are available. This invalidation causes the *main* instance to become invalid, since no other spell checker is available. This situation causes the 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 is re-validated, followed by the *main* instance, and then the application as a whole.

3.2. Approach limitations

The main limitation to this approach is the unpredictability of service dependency binding creation. Unpredictability results from the ambiguity that arises when there are multiple candidate services available to resolve a given service dependency. This situation occurs, for example, when a service dependency is singular, but at the time the instance manager tries to resolve the dependency, multiple candidates are present. Unpredictability can be reduced by using filters in required service interfaces, but this is not suffi-



a) Device monitoring application **b)** VersaTest extensible client

Fig. 3. Architecture of evaluation applications

cient. Knowing which of the available candidates is the best choice is difficult, if not impossible. This issue is similar to such research questions as locality, where the desire is to choose services that a physically near or appropriate.

4. Evaluations

This section describes two application scenarios in which the Service Binder[1] mechanism was used; the Service Binder, and the service-oriented component model as a whole, is built on top of the OSGi service platform [8]. The Service Binder simplifies the construction of applications on the OSGi services platform, where dynamic availability is typically handled programmatically.

4.1. Device monitoring at Schneider Electric

The Service Binder was used at Schneider Electric[2] 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 it must be possible to add or remove new monitoring components in the running system as new devices are connected or disconnected to or from the bus, respectively.

Figure 3a represents the architecture of the system. *Business objects* contain monitoring logic and provide a service that allows a s*cheduler* (the core in this application) to activate them periodically. Business objects have service dependencies on services that allow them to create logs and send e-mail notifications.

Support for the run-time addition or removal of business objects is achieved through the scheduler's service dependency on the `BOService`, which is optional, aggregate, and dynamic, and by the business objects' service dependency on the polling service. In this application, business objects and notification mechanisms can be continuously in-

1 Available for download at http://gravity.sf.net/servicebinder

2 http://www.schneiderelectric.com

troduced into or removed from the system as a result of continuous deployment activities that are triggered outside the application.

4.2. VersaTest client

The Service Binder was used at a company called Ascert, which creates a system for testing online transaction processing systems, called VersaTest[3]. VersaTest runs as a server and a client for the VersaTest server was built as an extensible environment where different client-side tools are integrated as plug-ins. The different tools communicate with the server and graphically display different types of information that result from the server's test cases. The VersaTest client is built around a core that provides the main functionality of the application, including a menu, toolbar, and a manager for multiple windows. These services allow multiple tools to work as a cohesive whole.

Figure 3b presents a simplified representation of the architecture of the VersaTest client system. The core component instance provides two services, one for adding entries to the menu and another one for creating windows. In this application, it is not the core component that requires services; instead, it provides services that are required by the tools. Tools not only require core services, but can themselves provide services and require services from other tools.

The VersaTest client application must support multiple configurations, where a configuration consists of the core and a particular set of tools. In this project, autonomous assembly capabilities provided by the Service Binder are leveraged as the VersaTest client is launched by starting an OSGi platform with a set of components corresponding to the different tools used in a configuration. This means that it is not necessary to create explicit compositions for each configuration, it is sufficient to simply include an arbitrary set of tools to compose. The Service Binder also allows tools to be added, removed, and updated during execution, but these capabilities are not currently used in the commercial version of the application.

5. Related work

Related work includes component models and service platforms, along with techniques to create auto-adaptive applications. Industrial component models include COM [2], and CCM [7]. Service platforms include OSGi, Jini, and web services. Jini [1] is a distributed Java-based service platform that introduces the concept of leases to support distributed garbage collection. Web services target business application interoperability. The OSGi's Wire Admin service [8] provides a mechanism to compose OSGi services between a producer and a consumer, but bindings are explicit and point-to-point.

Dynamically reconfigurable systems focus on changing the architecture of a system during execution. These systems use explicit architecture models and map changes in these models to the application implementation. Numerous works exist around self-adaptation through dynamic reconfiguration in component-based systems, such as [6] and [5]. This last work 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 to or removed from the

3 http://www.ascert.com/versatest.html

system according to constraints defined as an architectural style written in an ADL; however, the logic resulting from these constraints must be programmed.

6. Conclusions and future work

This paper described the experience of building component-oriented applications with a framework that supports run-time adaptation in response to the dynamic availability of functionality provided by constituent components. In this framework, each component instance is managed independently and its bindings are created and adapted at run time based on information associated with a component's service dependencies; instances are constantly managed to maintain their validity with respect to their service dependencies.

Applications built from this framework are capable of adding new functionality or releasing/substituting departing functionality. They also exhibit self-repairing characteristics since instance managers strive to maintain component instance validity and, consequently, the validity of the application as a whole. This framework was implemented on top of the OSGi framework and was successfully used in two different application scenarios.

Ongoing work includes studying mechanisms to reduce issues associated with unpredictability and ambiguity. These issues are exacerbated when multiple instances of the same application may exist at the same time, unlike the application scenarios described in this paper where only one application instance exists at any given time. An initial approach for addressing these issues is discussed in [4].

Finally, the authors would like to acknowledge the people at Schneider Electric and Ascert for their support and feedback.

References

[1] Arnold, K., O'Sullivan, R.; Scheifler, W. and Wollrath, A., "*The Jini Specification*," Addison-Wesley, Reading, Mass. 1999.
[2] Box, D., "Essential COM," Addison Wesley, 1998.
[3] Cervantes, H. and Hall, R.S., "Automating Service Dependency Management in a Service-Oriented Component Model," in Proceedings of CBSE 6, 2003
[4] Cervantes, H. and Hall, R.S., "Autonomous Adaptation to Dynamic Availability Through A Service-Oriented Component Model", ICSE 2004
[5] Georgiadis, I., Magee, J. and Kramer, J., "Self-organising software architectures for distributed systems," in Proceedings of the first workshop on Self-healing systems, 2002.
[6] Kon, F., "Automatic Configuration of Component-Based Distributed Systems," Doctoral dissertation, Department of Computer Science, University of Illinois. May 2000.
[7] Object Management Group. "CORBA Components: Joint Revised Submission," August 1999.
[8] Open Services Gateway Initiative. "OSGI Service Platform," Spec. Rel. 3.0, March 2003.
[9] Szyperski, C., "Component software: beyond object-oriented programming," ACM Press/Addison-Wesley Publishing Co., 1998.