Autonomous Adaptation to Dynamic Availability Using 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 project, called Gravity, that defines a component model, where components provide and require services (i.e., functionality) and all component interaction occurs via services. This approach introduces service-oriented concepts into a component model and execution environment. The goal is to support the construction and execution of component-based applications that are capable of autonomously adapting at run time due to the dynamic availability of the services provided by constituent components. In this component model the execution environment manages an application that is described as an abstract composition that can adapt and evolve at run time depending on available functionality. The motivation of Gravity is to simplify the construction of applications where dynamic availability arises, ranging from modern extensible systems to novel computing approaches, such as context-aware applications.

1. Introduction

This paper describes a project, called Gravity, that defines a component model, where components provide and require services (i.e., functionality) and all component interaction occurs via services. This approach introduces service-oriented concepts into a component model and execution environment. The goal is to support the construction and execution of component-based applications that are capable of autonomously adapting at run time due to the *dynamic availability* of the services provided by constituent components. Dynamic availability refers to a situation where services may become available or unavailable at any time during the execution of an application and this is outside of application control. An application must adapt at run time to these types of changes by looking for alternatives to departing services and integrating new services.

The Gravity project currently focuses on the construction of applications that are non-distributed and user-oriented (i.e., they support direct user interaction via a graphical user interface). User-oriented applications are directly impacted by dynamic availability. First, it is more and more common for user-oriented applications to be highly extensible and to act as a sort of *über-client*, which requires that the application incorporate new services for accessing new types of content and functionality. Applications, in this case, are in a constant state of assembly, continuously integrating new services as necessary. Second, a growing number of user-oriented applications are created using technologies, such as web services, where local components are stubs to remote entities that are inherently unreliable and likewise push this unreliability into applications in the form of dynamic availability. Examples of user-oriented applications that are impacted by dynamic availability range from extensible systems, such as web browsers or integrated development environments, to application managers, such as the ones found in desktop environments or personal digital assistants (PDAs).

The motivation for this research is to explicitly support dynamic availability of services in a component model and to simplify the creation of component-based applications that are capable of autonomously reacting to changes in service availability, such as a web browser that is capable of automatically incorporating plug-ins to visualize particular content. Another motivation is to propose mechanisms to support novel computing approaches, such as context-aware computing [14]. 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 automatically incorporate services obtained from the airport's wireless network, such as one that provides flight information. When the user leaves the airport, such services are no longer available.

The Gravity project is founded on the concept of a *service-oriented component model,* which adopts the service-oriented approach of late, non-explicit bindings among components via services. In this model, an application is described as an abstract composition that adapts and evolves at run time depending on service availability. A supporting execution environment manages application compositions as changes in service availability occur and provides continuous deployment and integration support.

Component and service orientation, which are the foundations for this work, are discussed and compared in the next section. Section 3 presents the principles of a service-oriented component model. Section 4 describes Gravity,

which is an execution environment that implements principles of a service-oriented component model. Section 5 presents an execution scenario, followed in section 6 by related work. Section 7 presents future work, followed by the conclusion.

2. Component and service orientation

This section describes the fundamental characteristics of both component and service orientation. A description of each approach is given followed by a comparison of the two approaches.

2.1. Component orientation

Component orientation promotes the construction of applications as compositions of reusable building blocks called *components*. Although there is no universal agreement on the definition for the term component, the definition formulated by Szyperski [26] 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."*

The component approach makes a distinction between component developers and assemblers, which are the third parties referenced in the definition above. Since component developers and assemblers may reside in different companies, it is necessary to deliver components as binary units to avoid source code release. A fundamental assumption of this approach is that the construction of an application is based on components that are physically available to the assemblers when composing (i.e., assembling) the application.

In the majority of industrial component models, such as EJB [28], CCM [22], and COM [4], components are similar to object-oriented classes in the sense that they can be instantiated and that their instances can be stateful; in this paper, the term "component" is used to indicate this meaning. A composition is created by an actor that instantiates some component instances, usually from a component factory, and then customizes and connects these instances to each other in some appropriate fashion.

To support composition, a component must provide information that describes the external structure of its instances, such as *provided* and *required* functional interfaces, and modifiable properties. The description of the external structure or *external view* is used by the application assembler to connect and customize the various component instances into a structural composition, similar to the approach of architecture definition languages (ADLs). Composition is performed using either a standard programming language or a specialized one. Hierarchical composition is achieved when the external view of a component is itself implemented by a composition.

Components are delivered and deployed independently as binary code along with their required resources, such as images and libraries. When installed, a component may require deployment dependencies be fulfilled before it can be instantiated. Finally, component instances require an execution environment that provides run-time support, normally through what is known as a *container* [10]. Run-time support includes life-cycle management and handling of *non-functional* characteristics such as remote communication, security, persistence, and transactions.

2.2. Service orientation

Service orientation shares the component-oriented idea that applications are assembled from reusable building blocks, but in this case the building blocks are *services*. A service is functionality that is contractually defined in a *service description*, which contains some combination of syntactic, semantic, and behavioral information. In service orientation, application assembly is based only on service descriptions; the actual *service providers* are located and integrated into the application later, usually prior to or during execution. As a result, service orientation focuses on "how services are described and organized in a way that supports the dynamic discovery of appropriate services at run time." [5]

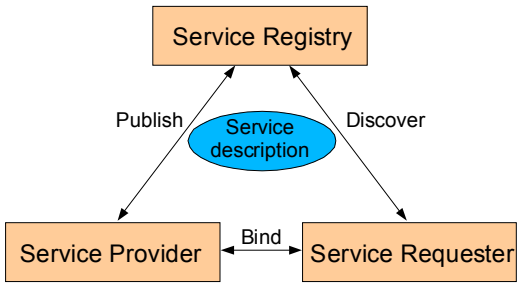

Figure 1. Service-oriented interaction pattern

To support dynamic discovery, service orientation is based on an interaction pattern involving three different actors; these actors are *service providers*, *service requesters*, and one or more *service registries* (see figure 1). Service providers publish their service descriptions into a service registry so that clients can locate them. Service requesters interrogate the service registry to find service providers for a particular service description. If suitable service providers are present in the service registry at the moment the request is made, the registry returns a reference to one or more service providers so that the service requester can select and bind to any of them. When the service requester binds to the service provider, the provider returns a reference to a *service object* that implements the service functionality.

Service orientation promotes the idea that a service requester is not tied to a particular provider; instead, service providers are substitutable as long as they comply with the contract imposed by the service description. Another funda-

mental characteristic of service orientation is that services exhibit dynamic availability, since they can be added or removed from the service registry at any time. Consequently, running applications must be capable of releasing departing services or of incorporating newly arriving services.

Since a service-oriented application is built by composing service descriptions, it is an abstract composition that only becomes concrete at run time. Service composition can be realized through standard programming languages, although a flow-based approach (e.g., processes) is favored in the domain of web services [11]. Hierarchical service composition is achieved when a service composition itself implements a service description.

Finally, a service-oriented platform provides infrastructure to support building applications according to service-orientation principles. In addition to a service registry, some service platforms provide notification mechanisms used to inform service requesters of the arrival or departure of services, others use the concept of a service lease, which means that service availability is guaranteed only for a determined time period after which the lease must be renewed.

2.3. Approach comparison

The two previous subsections present two different approaches for constructing applications from reusable building blocks. These two approaches promote reuse by creating a separation between building block development and application assembly.

There are, however, differences between these two approaches. One difference concerns the fact that in component orientation, applications are assembled from building blocks that are integrated at the moment of assembly, while in service orientation integration occurs prior to or during execution, since only service descriptions are available during assembly. As a consequence of this, service orientation places a stronger emphasis on service description and the separation between description and implementation; this enables better support for run-time discovery and substitution. Run-time component discovery and substitution are not primary concerns in component orientation, so a separation between the component's external view and its implementation is less strictly enforced.

A second difference is that building blocks exhibit dynamic availability in service orientation, since services can be registered or unregistered from the service registry at any time. This is not the case for component orientation, where the hypothesis that components may arrive or depart during execution is typically not supported. As a result, service compositions are more dynamic in nature than component compositions. Dynamism in service composition includes explicit support for incorporating new services that become available during execution or releasing departing services. Of course, a certain degree of dynamism is possible in component orientation, but in general this is not explicitly supported and must be done programmatically. Further, composition in component orientation is structural in nature, while service-oriented composition tends to focus on flow-based approaches, such as process-oriented composition.

A third difference is that the separation between component and instance is explicit in component orientation. This separation mandates that component instances be explicitly created. In service orientation, service object creation is implicit and not even considered by service requesters.

Finally, packaging is an important activity in component orientation, since a component is defined as an independently deliverable and deployable entity. Service orientation does not consider delivery and deployment; it is supposed that these activities occur prior to service registration and lookup.

3. Combining services and components

A service-oriented component model introduces concepts from service orientation into a component model. The motivation for such a combination emerges from the need to introduce *explicit* support for dynamic availability into a component model. This section first describes the principles of a service-oriented component model and then discusses the challenges of implementing one.

3.1. Principles

A service-oriented component model is based on the following principles:

- *A service is provided functionality.* A service is a reusable operation or set of operations.
- *A service is characterized by a contract.* A contract defines a service's characteristics for composition, interaction, and discovery. This is achieved by describing some combination of the service's syntax, behavior, and semantics. An important syntactical aspect of a contract is that it explicitly declares a service's dependencies on other services to enable structural composition.
- *Components implement a contract.* By implementing a contract, the component provides the described service and obeys its constraints. In addition to the service dependencies specified in the contract, a component may also declare additional implementation-specific service dependencies. Services are the sole means of interaction among component instances.
- *The service-oriented interaction pattern is used to resolve service dependencies.* Services provided by component instances are published into a service registry. The service registry is used to dynamically discover services at run time for resolving service dependencies.
- *Compositions are described in terms of contracts*. A composition is a set of contracts that are used to select concrete component to instantiate. Explicit bindings are not necessary, since they are inferred at run time from the service dependencies declared in the con-

stituent contracts. Compositions become concrete at run time as components that provide the constituent contracts are discovered and instantiated into the composition. Composition is a continuous run-time activity that responds to the availability of services.

- *Contracts are the basis for substitutability*. In a composition, any component that implements a given contract can be substituted with any alternative component that implements the same contract.

These principles enable explicit support for dynamic availability of services for a variety of reasons. First, it is not necessary to select particular components for a composition, since compositions are defined in terms of contracts, which are a basis for substitutability. Further, substitutability is strengthened by recognizing two levels of dependencies, both contractual and implementation specific; this is different from both traditional component and service orientations. In component orientation, only implementation dependencies exist, which hinders substitutability. In service orientation, service dependencies are not declared as part of the contract, which eliminates the possibility of structural composition. Dynamic availability is also enabled because explicit bindings are not necessary in a composition, since they are inferred from the contracts. Finally, all service dependencies are resolved at run time using the service-oriented interaction pattern and this interaction pattern forms the basis of a continual composition life cycle.

3.2. Challenges

There are two main challenges in creating a service-oriented component model; these are dealing with ambiguity and dynamic availability.

Ambiguity is inherent to service orientation and arises when multiple candidate services are available that can resolve a given service dependency. In such a situation, it is difficult to know which of the candidate services is the best one to choose. Ambiguity is related to other research questions, such as locality, that involve choosing the best option among many valid choices. In a given situation, the best choice may be one that is physically near, inexpensive, secure, or any other metric that is important to the application. There is no single or complete way to deal with such ambiguities and only partial solutions are possible. One approach for limiting ambiguity is through rich discovery mechanisms that use semantics and behavior, in addition to syntax, to reduce the number of candidate services for resolving a given service dependency.

Dynamic availability of services results for any number of reasons, including sophisticated concepts such as context awareness or fundamental concepts related to continuous deployment activities. Since these changes occur at run time, they have repercussions on existing component instances and the compositions in which they reside. Addressing dynamic availability is a complex task that requires component instances to listen for service departure or arrival and applications to listen for component departure or arrival. In response to changes in service availability, component instances must adapt by searching for alternative services, for example. Applications must adapt by finding replacements for departing components or by incorporating instances of newly arriving components.

Gravity, which is discussed in the next section, focuses on supporting dynamic availability, although it currently provides only simple mechanisms to deal with ambiguity.

4. Gravity

Gravity implements principles of a service-oriented component model and also provides a Java-based execution environment that manages run-time changes that result from dynamic availability. The benefit of this approach is that dynamic availability is supported explicitly and that adaptation logic is handled by the execution environment, which simplifies application development. To achieve this, Gravity adopts a two-level management approach, where "local" constraints are managed at the component instance level and "global" constraints are managed at the composition level. These topics are presented in the following subsections; they are illustrated with a simple example that depicts the construction of a graphical web browser that is extendable using plug-ins.

4.1. Support for Dynamic Availability

The Gravity execution environment explicitly manages dynamic availability of services for applications. The following scenarios are supported by Gravity:

- *Arrival of a new component.* An existing composition may wish to integrate or substitute an existing component instance with one created from the newly arriving component; this situation can occur for different reasons. For example, a composition may be able to accommodate an indefinite number of instances that provide a particular service, such as an application that can be extended by plug-ins, or a composition may wish to substitute an existing instance by one created from the new component because its service offers some better characteristic.
- *Departure of an existing component.* All of the instances created from the departing component also need to be removed from compositions in which they reside. This is necessary since the instances are likely to be in an unusable state. For example, a component may provide a Bluetooth print service, which has a dependency on a physical printer; when the printer is no longer in range, all of its instances cease functioning.
- *Arrival of a new component instance.* A newly introduced instance's provided services, if any, are published into the service registry.
- *Departure of an existing component instance.* A removed instance has its provided services, if any, removed from the service registry.

- *Arrival of a new service.* Existing component instances may use the new services to fulfill existing service requirements. As a consequence, previously unavailable services provided by these newly fulfilled instances become available.
- *Departure of an existing service.* Existing component instances that depend on the departing service may be affected by its removal from the service registry depending on the characteristics of their dependencies. As a consequence, their services may also be unregistered.

To manage dynamic availability, Gravity follows a two-level approach that manages instances at a local level and compositions at a global level.

4.2. Instance-level management

Instance-level management is "local," since it only concerns service dependency management for a particular component instance independent of the application in which it resides. In Gravity, component instances can have one of two states: *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. Component instances in Gravity are "intentional," since a request to create an instance initially starts out invalid and then may alternate between valid and invalid states as its service dependencies are dynamically satisfied and unsatisfied, respectively. This occurs until the instance is explicitly destroyed. A component description provides Gravity with the necessary information to manage service dependencies for component instances.

4.2.1. Component description. In the Gravity implementation, a component description represents a contract, similar to the contract concept defined in section 3.1, although it does not currently separate the service contract from the component implementation details; this is the subject of future work. The component description contains a list of provided services, a set of service properties (whose purpose is to identify the component instance from other providers of the same services), and a set of service dependencies that need to be fulfilled for a component instance to become valid. Additionally, the component description includes the name of a class that implements the component. A component description may describe a singleton instance or a factory of instances. In the case of a singleton instance, the component instance is automatically created when the component is deployed. In the case of a factory, component deployment results in the automatic registration of a special factory service, which is used to create and destroy instances of the component.

Service dependencies have properties that determine how bindings between an instance and the services it requires are created and managed at run time. Service dependencies are characterized by a fully qualified service name and three properties: *cardinality*, *binding policy*, and an optional *filter*.

```
<component class="org.gravity.webbrowser.BrowserImpl"
           factory="yes">
  <provides
    service="org.gravity.services.Application"/>
  <provides service="org.gravity.services.WebBrowser"/>
  <property name="version" value="1.0" type="string"/>
  <requires
      service="org.gravity.services.BrowserPlugin"
      filter=""
      cardinality="0..n"
      policy="dynamic"
      bind-method="addPlugin"
      unbind-method="removePlugin"
    />
  <requires
      service="org.gravity.services.WindowManager"
      filter=""
      cardinality="1..1"
      policy="dynamic"
      bind-method="setWindowManager"
      unbind-method="unsetWindowManager"
    />
</component>
```

Figure 2. A component descriptor

Cardinality is used for expressing both optionality and aggregation. Optionality refers to whether the service dependency represents a mandatory or non-mandatory binding, while aggregation refers to whether the service dependency represents single or multiple bindings. 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.

Binding policy determines how dynamic service changes are handled: a *static* binding policy indicates that bindings cannot change at run time without causing the instance to become invalid (at which point its services are unregistered), whereas a *dynamic* binding policy indicates that bindings can change at run time as long as bindings for mandatory dependencies are satisfied.

Service dependencies may also include a filter (in LDAP query syntax) that evaluates over the service properties attached to service providers when their services are registered. This filter is used to limit ambiguity by reducing the number of potential service providers that can be used to satisfy the associated service dependency.

Figure 2 shows an example of a component descriptor. In this example, the component provides both a `WebBrowser` service and an `Application` service. The component has a service dependency on a `BrowserPlugin` service. This dependency is characterized as dynamic and has a cardinality of `0..n`. In this case, the cardinality means that multiple `BrowserPlugin` services can be connected to an instance of the component and that plug-in services are optional. The component also has a service dependency on a single `WindowManager` service that is required to create windows.

4.2.2. Run-time management. In Gravity, instances are managed by the execution environment using a mechanism named the Service Binder [7]. During execution, each component instance is managed by an *instance manager* that is responsible for creating bindings between the instance it manages and other instances that provide required services. The instance manager also controls the instance's life cycle, manages service registrations, and maintains the instance's validity. Bindings between component instances are created following the service-orientation interaction pattern, where the instance manager queries the service registry to find services provided by other instances. Binding management, which includes creation or destruction of bindings, occurs when the instance manager receives notifications announcing changes in the service registry. The instance manager is similar to a container that extracts adaptation logic from the components and that implements the *inversion of control* pattern [15]. The component class implements service binding and unbinding methods that the instance manager uses to set service references. Additionally, the component class may implement control interfaces that are discovered dynamically and allow component instances to participate in dynamic reconfiguration activities.

This approach for instance-level management allows an application to be constructed as a set of interconnected component instances that self-assemble and self-adapt with respect to dynamic availability. More precisely, an application built from these mechanism can adapt with respect to the addition, removal, and substitution of a component instance [8]. However, instance-level management does not have a global view of an application, which must be addressed by composition-level management.

4.3. Composition-level management

Instance-level management is only concerned with composition at a "local" level, since an instance manager is only responsible for managing a single instance. An instance manager does not possess a global view of the application. While this is sufficient for a certain class of applications (see section 7), constructing applications often requires a global view. This global view, which is described at the composition level, contains information about the services that are necessary to build a particular application. Composition-level management guides instance creation and also manages changes resulting from dynamic availability of services. Composition-level management must also address issue of unpredictability.

4.3.1. Unpredictability and composition scopes. Instance-level management faces the problem of *unpredictability* due to ambiguity. If an instance has a singular dependency on a particular service and two providers of the service are present at a given moment, the choice of the provider that is used cannot be predicted. Unpredictability can be reduced by using filters in required service interfaces, but filters have two downsides. The first is that a filter

```
<composition factory="yes">
  <exports from="core"
    service="org.gravity.services.Application"
    type="provided" />
  <exports from="core"
    service="org.gravity.services.WindowManager"
    type="required" />
  <property name="appName" value="WebBrowser"
    type="string"/>
  <placeholder id="core" filter="" cardinality="1..1">
    <provides
      service="org.gravity.services.Application"/>
    <provides
      service="org.gravity.services.WebBrowser"/>
    <requires
      service="org.gravity.services.WindowManager"
      filter=""
      cardinality="1..1"
      policy="dynamic"/>
    <requires
      service="org.gravity.services.BrowserPlugin"
      filter=""
      cardinality="0..n"
      policy="dynamic"/>
  </placeholder>
  <placeholder id="plug" filter="" cardinality="0..n">
    <provides
      service="org.gravity.services.BrowserPlugin"/>
  </placeholder>
</composition>
```

Figure 3. A composition description

that is too strict excessively reduces the possibility of service provider substitution. The second is that filters are not sufficient to avoid ambiguity when multiple instances of the same component exist, since provided services from these instances look identical to each other in the service registry. Composition *scopes* are introduced to reduce these issues.

Scopes constrain dependency management inside a well defined boundary. Upon creation, component instances are placed inside a particular scope. Each scope contains an independent service registry that is used by instance managers inside the scope to manage dependencies; this means that services in one scope are not visible to an instance manager in another scope. Scopes are hierarchical, where the root of the hierarchy is called the global scope; this scope hosts singleton instances.

Services that are provided or required by an instance inside a scope may be *exported* to the parent scope. Since every scope corresponds to a particular composition, and scopes can be nested, this enables hierarchical compositions. Scopes can be created or destroyed dynamically. The destruction of a scope results in the destruction of the instances and child scopes that it contains.

4.3.2. Composition description. A composition description defines sets of provided and required services, called *placeholders*. Each placeholder represents a contract, similar to the contract concept defined in section 3.1, that guides component selection and the creation of one or more component instances into the composition scope. The placeholder is characterized by a unique identifier, a filter, and a cardinality. The filter is used to select among multiple components and the cardinality defines whether a placeholder represents an optional or mandatory instance and if it repre-

sents a single or aggregate instances.

Gravity's service-oriented component model supports the concept of hierarchical composition by allowing a composition description to be used as a component. Hierarchical composition is achieved by allowing compositions to export provided and required services. Such compositions can be created using a factory and integrated into other compositions. To create a factory for a composition, the composition description must include the `factory` attribute in the composition tag. To export provided or required services from a placeholder, the `exports` tag is used, which references placeholders inside the composition. To avoid changes in the external view of the component at run time, exports can only reference placeholders that are mandatory single instances (that is with a cardinality of `1..1`); lifting this restriction is the subject of future work. Additionally, service properties can be attached to the composition.

Figure 3 shows a simple composition based on two placeholders, one that represents a component that provides the `WebBrowser` and `Application` services and one that provides a `BrowserPlugin` service. This composition represents a web browser that can incorporate multiple plug-ins at run time; plug-ins in this case are optional. Since this composition is a factory, multiple instances of it can be created. Each composition instance is created in a different scope, which means that plugin instances are not shared among browsers.

4.3.3. Run-time management. As with instance-level management, compositions are also valid or invalid. When a composition is valid, its associated scope is created and services from its instances are exported. When a composition is invalid, the scope associated with the composition is destroyed.

Composition creation. Every composition is managed by a composition manager that first locates appropriate components for each placeholder declared in the composition description. Components are located through `Factory` services and selected according to the external view and a filter that references the properties associated with a component. The criteria used to select a component for use in a composition is that it must, at minimum, match the external view defined in the placeholder description. The component may provide additional services and require additional services, but the latter must be optional. The fact that additional required services need to be optional is to guarantee that the instance can eventually become valid inside the composition. When factories for every mandatory instance are located, the scope associated with the composition is created and instances are created and introduced inside the scope. From that moment instance-level management automatically composes the constituent components.

Composition management. As changes in the services occur due to dynamic availability, some instances may need to be removed or added to a composition. The consequences that these changes have on a valid composition depend on the cardinalities associated with placeholders. The departure of an instance from an optional placeholder does not invalidate the composition. The departure of an instance from a mandatory placeholder invalidates the composition if the composition manager is not capable of locating a component that can provide a replacing instance. New instances may be incorporated at any time through placeholders that represent aggregate instances. The composition manager from the execution environment constantly strives to maintain the validity of a composition.

4.4. Other run-time support

The Gravity execution environment provides support for continuous component deployment; this includes install, activation, update, and removal of components and compositions. Components and compositions are packaged as JAR files and installed from URLs. When the execution environment activates a component, it triggers factory registration or singleton instance creation; however, this is only possible after component deployment dependencies are fulfilled. Deployment dependencies relate to required or provided Java packages and are declared inside a manifest in the component JAR file. Deployment dependencies are managed automatically by the execution environment.

Gravity also provides a design environment [17], where component compositions are created visually using drag-and-drop gestures from a list of factories onto a design canvas. Once created, instances can be visually customized. The design environment allows the end user to switch from design mode to run-time mode and additionally provides simple mechanisms to support changes in the user interface due to changes in component availability during execution.

5. Execution scenario

This execution scenario describes an application launcher that allows the end user to create instances of components that provide an `Application` service. The application launcher component provides a `WindowManager` service and has an aggregate service dependency on `Factory` services for components that provide an `Application` service. Since the application launcher's dependency on factories is an aggregation, the application launcher is automatically bound to `Application` factories as they appear in the execution environment. For this scenario, the web browser component depicted in figure 2 and the web browser composition depicted in figure 3 have been deployed into the execution environment. The application launcher provides a menu of `Application` factories, which allows the end user to create application instances.

When the end user selects an entry from the factory menu, the corresponding component factory is used to create a component instance that provides an `Application`. In this scenario, the only available application is a simple web browser, defined by the aforementioned composition, which is created inside the application launcher's window when launched; figure 4a shows two instances of

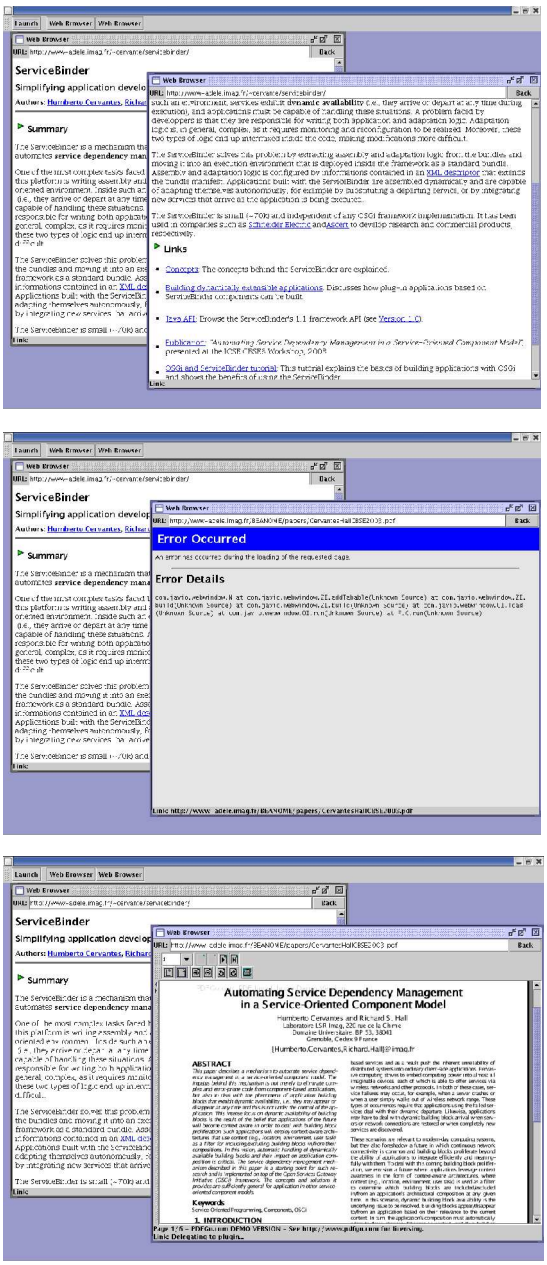

Figure 4. Application Launcher and Web Browser

the web browser composition. Initially, no components that offer `BrowserPlugin` services are available in the execution environment, so when the end user selects a link leading to content requiring the use of a plugin (in this case a PDF file), an error is displayed and is depicted in figure 4b. If a component providing a `BrowserPlugin` service to display PDF content is installed in the execution environment, then instances of this plug-in component are automatically created and integrated into each web browser composition scope. Now if the end user selects a link pointing to a PDF file, the contents are correctly displayed, as depicted in figure 4c. If the plug-in component is removed, browsers that are displaying a PDF file at that moment will modify their display to reflect the change.

Although this execution scenario may appear simple, the main purpose is to demonstrate how Gravity simplifies the construction of an application that supports dynamic availability of its constituent services. With these mechanisms it is possible to create applications that exhibit sophisticated auto-adaptive behavior.

6. Related work

Component models including COM [4], JavaBeans [27], EJB [28] and CCM [22] are based on the concepts described in section 2.1, which are also shared by Gravity's service-oriented component model. These component models target various application domains, but none explicitly supports dynamic availability other than via programmatic methods. These models also follow a programmatic approach toward component composition (except CCM) and do not explicitly support hierarchical composition. EJB and CCM support non-functional aspects such as persistence, transactions, and distribution, which are not currently supported in Gravity. Gravity's support for dynamic discovery of control interfaces (see section 4.2.2) could be used as a starting point to implement some of these features.

Service-oriented platforms include OSGi [23], Jini [2], web services [11] and Avalon [1]. OSGi defines a framework to deploy services in a centralized environment, but leaves service dependency management to programmers. OSGi is used as a substrate on top of which the Gravity execution environment is built. Jini is a Java-based distributed architecture that supports the existence of multiple service registries. Jini introduces the concept of service leasing as a mechanism that limits the time a client can access a service. Web services target business application interoperability and are built upon XML-based protocols for service description (WSDL) and communication (SOAP). Web services are registered in a service registry called UDDI. Composition in web services is achieved mostly by use of flow-based models, as in BPEL4WS [12]. Web services relationship to Gravity is mostly conceptual, although Gravity components may act as stubs to web services. Avalon has concepts similar to a service-oriented component model, since Avalon components are connected following a service-orientation interaction pattern; dynamic availability is not supported by this platform. All these service platforms do not support automated service dependency management.

Composition languages [21] leverage ADL concepts (architectural description targeted towards documentation and analysis) to define component compositions that include scripts, coordination primitives, and adaptation mechanisms. Script execution performs tasks such as component creation and wiring, but is not oriented toward supporting run-time dynamic changes. The CoML composition language [3] is related to Gravity since it defines abstract compositions, in this language, however, compositions become

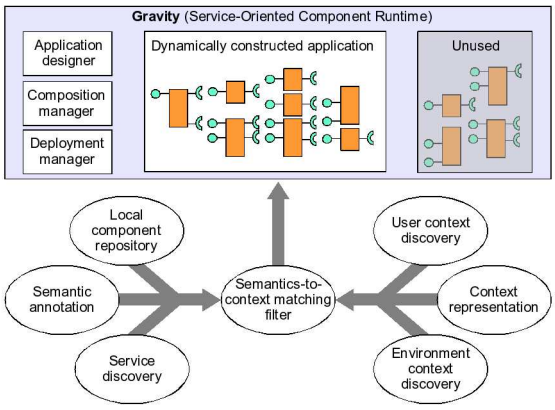

Figure 5. A filter directs dynamic availability

concrete at design time and not at run time.

Dynamically reconfigurable systems focus on changing the architecture of a system during execution [18]. These systems use explicit architecture models and map changes in these models to the application implementation [24]. Self-adaptation through dynamic reconfiguration in component-based systems is studied in [19] and [16], among others. This last work explores the possibility of constructing self-assembling and self-adaptive applications based on architectural constraints, however, their approach requires that adaptation logic is written programmatically.

Component-based development environments, such as IBM's Eclipse [25] and WREN [20], are related to the design environment provided by Gravity, but these environments assume that composition cannot occur during application execution. Sun's Bean Builder [13] supports switching between design and run-time modes, but is not built upon a component model where dynamic availability is present.

7. Validations and project status

A Gravity prototype and demonstration is available, as well as an open source implementation of the Service Binder technology mentioned in this paper.[1] The Service Binder technology has been used in at least two external projects [8]. One project is an industrial research prototype that deals with the management of electric devices connected through a bus to an OSGi framework. Devices can be added or removed at any moment and this requires that components be added or removed dynamically from the execution environment. The second project is a commercial application that is an extensible client for monitoring an online transaction processing system. The client can be extended through various tools that provide different views of the system that is being monitored. In this project, the automatic dependency binding is used to dynamically assemble the application out of arbitrary configurations of deployed components. These projects are based on systems constructed exclusively from singleton instances. Support for composition management has only recently been included in the Service Binder and external evaluations for it are expected to occur in the near future.

1 available at http://gravity.sf.net/

8. Future work

Gravity currently solves many pragmatic issues with respect to supporting dynamic service availability and it provides an effective platform for further experimentation. This section discusses important open research questions in no particular order.

Service description. The current Gravity implementation does not separate service description from component description; this issue must be addressed in a next version. Additionally, service semantics and behavior are currently determined through fully qualified service interface names. One approach is to annotate services with semantic description, as in [9]. Better service description is needed to improve service discovery and selection, which further addresses issues related to ambiguity.

Supporting context awareness. One of the motivations behind Gravity is to create an infrastructure to support novel computing approaches such as context awareness. In this approach, contextual information, such as the user location, instigates changes in the behavior and functionalities provided by an application.

The vision of how Gravity could be used in supporting context awareness is depicted in figure 5. In this figure, a *filter* guides composition creation and component deployment in an execution environment based on contextual information coming from different sources. Depending on the information, the filter decides which components are installed or removed from the execution environment during application execution. Contextual changes become the source of dynamic availability of services. Applications that are executing inside the platform adjust autonomously to the changes in services using the mechanisms described in this paper. A benefit of this approach is that it promotes a separation between contextual information processing and the infrastructure that manages the changes in an application resulting from this information.

Flexible user interfaces. Since user-oriented applications support interaction via user interfaces, the consequences of dynamic availability at the user interface level must be studied. Research in plastic [6] user interfaces is necessary to make dynamic changes to the user interface smooth and unobtrusive.

State transfer. If an instance must be replaced inside of a composition due to dynamic availability issues, its state could be transferred to the incoming instance to lessen the impact of service loss. While this is not currently implemented in Gravity, the control interfaces mechanism discussed in section 4.2.2 could be used as a way to implement this functionality.

Distributed architecture. Gravity currently studies the mix between component and service orientation in a centralized environment; this approach could, however, be applied to distributed environments too.

9. Conclusions

This paper describes a project, called Gravity, with the goal to support the construction and execution of component-based applications that are capable of autonomously adapting at run time to the *dynamic availability* of the services provided by constituent components.

Gravity's approach introduces support for dynamic availability into its component model by adopting concepts from service orientation. To this end, Gravity defines the concept of a service-oriented component model and provides an execution environment to support applications built with it. The service-oriented component model is based on the idea that component instances are service providers that may register and/or unregister their services at any time. Services provided by component instances are discovered at run time using the service-oriented interaction pattern as components and their instances are introduced into the execution environment. Dynamic availability is managed by an execution environment, built on top of the OSGi service platform, that extracts adaptation logic from components and applications.

While Gravity simplifies the construction of modern applications where dynamic availability occurs, such as extensible systems, it was conceived as a way to support novel computing approaches, such as context awareness. An essential element of Gravity's execution environment, the Service Binder, has been used and evaluated in an industrial research prototype and a commercial application. Feedback from these projects has been very encouraging.

10. References

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

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

[3] Birngruber, D., "*CoML: Yet Another, But Simple Component Composition Language*," Proceedings of the Workshop on Composition Languages, 2001

[4] Box, D., "*Essential COM*," Addison Wesley, 1998.

[5] Burbeck, S., "*Evolution of Web Applications into Service-Oriented Components with Web Services*," Online Document, October 2000. (http://www-106.ibm.com/developerworks/library/ws-tao/)

[6] Calvary, G., Coutaz, J., and Thevenin, D., "*Supporting Context Changes for Plastic User Interfaces: A Process and a Mechanism*," Proc. of IHM-HCI, 2001.

[7] Cervantes, H. and Hall, R.S., "*Automating Service Dependency Management in a Service-Oriented Component Model,*" Proceedings of the 6th Workshop on Component-Based Software Engineering, May 2003.

[8] Cervantes, H. and Hall, R.S., "*A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences*," To appear in the 7th Symposium on Component-Based Software Engineering, May 2004

[9] Chakraborty, D., Perich, F., Avancha, S. and Joshi, A., "*DReggie: Semantic Service Discovery for M-Commerce Applications*," in Workshop on Reliable and Secure Applications in Mobile Environments, 2001.

[10] Conan, D., Putrycz, E., Farcet, N. and DeMiguel, M., "*Integration of Non-Functional Properties in Containers*," in 6th Workshop on Component-Oriented Programming, 2001.

[11] Curbera, F., Nagy, A. and Weerawarana, S., "*Web-Services: Why and How,*" in Workshop on Object-Oriented Web Services (in OOPSLA), August 2001.

[12] Curbera, F. et Al., "*Business Process Execution Language (BPEL) for Web Services, Version 1.0*," IBM, July 2002.

[13] Davidson, M., "*The Bean Builder Tutorial*," Online Document, 2002. http://java.sun.com/beans

[14] Dey, A. and Abouwd, G., "*Towards a Better Understanding of Context and Context-Awareness*," Workshop on the What, Who, Where, When and How of Context Awareness at CHI, 2000.

[15] Fowler, M., "*Inversion of Control and the Dependency Injection Pattern*," Online Document, 2004. (http://martinfowler.com/articles/injection.html)

[16] Georgiadis, I., Magee, J. and Kramer, J., "*Self-organising software architectures for distributed systems*," in the First Workshop on Self-Healing Systems, 2002.

[17] Hall, R.S. and Cervantes, H., "*Gravity: Supporting Dynamically Available Services in Client-Side Applications*," poster paper at ESEC/FSE 2003.

[18] Hofmeister, C.R., "*Dynamic Reconfiguration of Distributed Applications*," Ph.D. Thesis, Computer Science Department, U. of Maryland, 1993.

[19] Kon, F. and Campbell, R.H. , "*Supporting automatic configuration of component-based distributed systems*," in Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, 1999.

[20] Lüer, C. and Rosenblum, D.S., "*WREN - An Environment for Component-Based Development*," published as Software Engineering Notes 26, 5, 2001.

[21] Nierstrasz, O., and Meijler, T.D., "*Requirements for a Composition Language*," Object-Based Models and Langages for Concurrent Systems, Springer-Verlag, 1995.

[22] Object Management Group. "*CORBA Components: Joint Revised Submission*," August 1999.

[23] Open Services Gateway Initiative. "*OSGI Service Platform*," Specification Release 3, March 2003.

[24] Oreizy, P and Taylor, R.N., "*On the Role of Software Architectures in Runtime System Reconfiguration*," IEEE Software, vol 145, no. 5, Oct. 1998.

[25] OTI Inc. "*Eclipse Platform Technical Overview*," http://www.eclipse.org/whitepapers/eclipse-overview.pdf, 2003.

[26] Szyperski, C., "*Component software: beyond object-oriented programming*," ACM Press/Addison-Wesley Publishing Co., 1998.

[27] Sun Microsystems. "*JavaBeans Specification*," Version 1.0.1, 1997.

[28] Sun Microsystems. "*Enterprise JavaBeans Specification*," Version 2.0, 2001.