Gravity: Supporting Dynamically Available Services in Client-Side Applications

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

ABSTRACT

This paper describes a project, called Gravity, that is providing support for building client-side applications out of dynamically available building blocks. The purpose behind this work is not only to deal with real-world issues already facing developers and end-users, but to also work toward a grander vision. In this vision, applications are built using context-aware architectures, meaning that context (e.g., location, environment, user task) is used as a filter to determine which building blocks are relevant to the application at any given time. The main concept underlying this vision is *dynamically available* building blocks, i.e., building blocks that can appear or disappear at any time. The Gravity technology described in this paper is a starting point for such research.

Categories and Subject Descriptors

D.2.6 **[Software Engineering]**: Programming Environments – *graphical environments*; D.2.11 **[Software Engineering]**: Software Architectures – *languages*; K.6.3 **[Management of Computing and Information Systems]**: Software Management – *software development*.

General Terms

Design

Keywords

Component-Oriented Programming, Service-Oriented Programming, OSGi, Dynamic Availability

1 INTRODUCTION

This paper describes a research project, called Gravity, that envisions a future where all applications are built from re-usable building blocks, such as components and web services. This future leads to the proliferation of building blocks beyond software developers' ability to integrate them into applications efficiently or effectively. This situation is further exacerbated by pervasive computing and ubiquitous network connectivity where literally all devices offer services for dynamic integration into client applications.

In response to this, Gravity pushes a vision of client-side applications that easily and inexpensively undergo continual evolution and adaptation by using context (e.g., location, environment, user task) to dynamically filter available building blocks. Unlike current client-side technology, which provides for limited, semi-static forms of change, such as the occasional update from the software vendor or the rigid extension mechanism of the plug-in, the goal of Gravity is to enable client-side applications to evolve and adapt dynamically with respect to virtually any and all changes in their design, deployment, and usage, and to do so as a normal and seamless part of their execution behavior. To do this, Gravity currently focuses on one significant underlying assumption: that application building blocks exhibit *dynamic availability*. Specifically, this refers to the situation where application building blocks appear or disappear at anytime and this cannot be controlled by the application.

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

This paper describes the initial steps Gravity has taken to address issues of dynamic availability of building blocks by combining

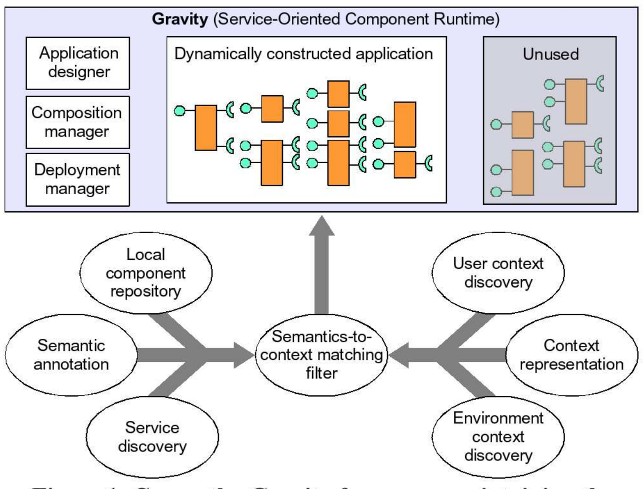

Gravity (Service-Oriented Component Runtime)
Application designer
Composition manager
Deployment manager
Dynamically constructed application
Unused
Local component repository
Semantic annotation
Service discovery
Semantics-to-context matching filter
User context discovery
Context representation
Environment context discovery

Figure 1: Currently, Gravity focuses on maintaining the application's compositional consistency in response to changes in the availability of components.

component-oriented and service-oriented concepts into an application framework.

2 APPROACH

Gravity views component orientation and service orientation as complementary. Component orientation focuses on composition and independent deployment, while service orientation focuses on description, discovery, and dynamic integration. Current work in Gravity investigates simplifying building and using applications whose building blocks may appear or disappear at anytime. Figure 1 depicts the focus of the current work (along the top of the figure) and its relationship to the overall vision. The Gravity prototype is implemented on top of the Open Services Gateway Initiative services gateway platform [9].

2.1 Service-Oriented Component Model

The notion of a *service-oriented component model* arises from the concepts of *service-oriented programming (SOP)*. In SOP, a service is a contract of defined behavior and semantics. A service client is not tied to a particular service provider, instead, service providers are interchangeable [1]. Service-oriented solutions follow a pattern that consists of *service providers*, *service requesters*, and a *service registry*. With respect to Gravity, a component model is defined as "service oriented" if it provides a service registry for publishing and discovering services that are offered by component instances.

In Gravity, *components* are black boxes that provide specific implementations of services and may also use services implemented by other components. A *service* is simply a Java interface. The interface itself is not used to derive the associated semantics of the service; instead, service interfaces and their semantics are defined externally, such as in a specification document, and any implementer of the service interface guarantees to faithfully implement the semantics of the service specification. Gravity's *service registry* contains references to published services that are implemented (i.e., provided) by component instances. The service registry is globally accessible to all component instances for purposes of discovering available services. The service registry allows services to be registered with an associated set of properties (i.e., attribute-value pairs). Components search for available services by performing a query over the associated properties and the desired service interface name.

2.2 Supporting Framework

Gravity's service-oriented component model does not, in and of itself, simplify the complex tasks of dealing with dynamically available building blocks; the Gravity framework plays a major role.

Component Deployment. Gravity stresses the concept of "*deploy at any time*," since applications supporting dynamic building block availability are always in a state of deployment. This means that deployment activities are also possible at run time. The Gravity framework supports installation, update, activation, and removal of components.

Application Design. Similarly to deployment, Gravity stresses the concept of "*design at any time*," since an application's design must change in response to building block availability. For client-side applications, the framework supports two kinds of design: architectural (or compositional) and aesthetic. Architectural design pertains to the actual "node and arc" concepts of software architecture, whereas aesthetic design pertains to graphical user interface composition and layout. The framework provides both automated and user-directed support for both of these design areas. Design support is available all the time, since it is not possible to control when new building blocks arrive or depart.

Service Dependency Management. Gravity simplifies support for dynamic building block availability by introducing the *Service Binder* as a mechanism to automate service dependency management in its component model. Using the Service Binder, a developer only provides dependency meta-data about his components and the instances he wants to create, instead of writing complex and error-prone service management code. The meta-data used by the Service Binder is contained in an XML file, called an *instance descriptor*, which is essentially of list of component types and instances to create. The instance descriptor file is placed in the component deployment unit (i.e., JAR file).

For each component instance described in the instance descriptor, the Service Binder creates an *instance manager*. The instance manager has four responsibilities:

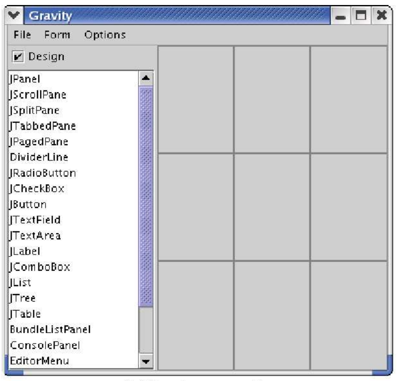
- dynamically monitor the component instance's service dependencies,
- create/destroy the component instance when its service dependencies are satisfied/unsatisfied,
- bind/unbind required services to/from the component instance when it is created/destroyed, and
- register/unregister any services provided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

Each instance manager actually represents the *intention* of creating a component instance and each tries to constantly maintain this intention throughout its lifetime, until it is explicitly disposed.

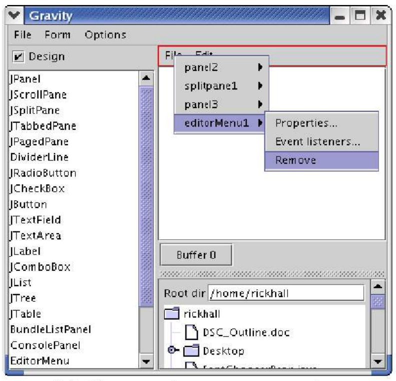
The description of an instance, inside the instance descriptor, includes the name of the class for the component, the set of services implemented by the component, the set of properties associated with the services, and a set of service dependencies for the component instance. Service dependencies are characterized by the fully qualified service interface name, a property filter, *cardinality*, and *binding policy*.

Cardinality is used to express optionality, such as a zero-to-one dependency, and also to express aggregation, such as a one-to-many dependency. Binding policy is specified as either *static* or *dynamic* and determines how run-time service changes are handled and how the component instance life cycle is managed. A static binding policy indicates that dependency bindings cannot change at run time without invalidating the associated instance, whereas a dynamic binding policy indicates that dependency bindings can change at run time.

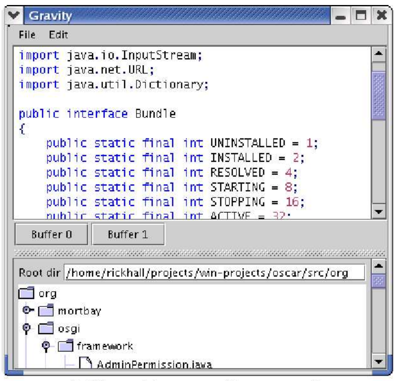
Despite the relative simplicity of the meta-data, applications using the Service Binder exhibit interesting auto-adaptive characteristics. For example, it is easy to describe a dynamic plugin-oriented system, such as a web browser, using a zero-to-many dynamic dependency between the browser and plugin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



a) Design mode.



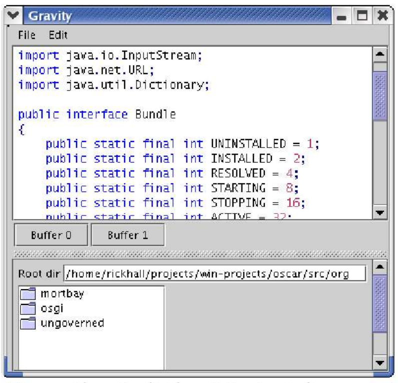
b) Composing components.



```
import java.io.InputStream;
import java.net.URL;
import java.util.Dictionary;

public interface Bundle
{
    public static final int UNINSTALLED = 1;
    public static final int INSTALLED = 2;
    public static final int RESOLVED = 4;
    public static final int STARTING = 8;
    public static final int STOPPING = 16;
    public static final int ACTIVE = 32;
```

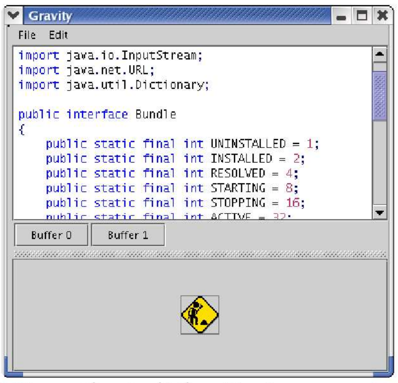
c) Gravity run-time mode.



```
import java.io.InputStream;
import java.net.URL;
import java.util.Dictionary;

public interface Bundle
{
    public static final int UNINSTALLED = 1;
    public static final int INSTALLED = 2;
    public static final int RESOLVED = 4;
    public static final int STARTING = 8;
    public static final int STOPPING = 16;
    public static final int ACTIVE = 32;
```

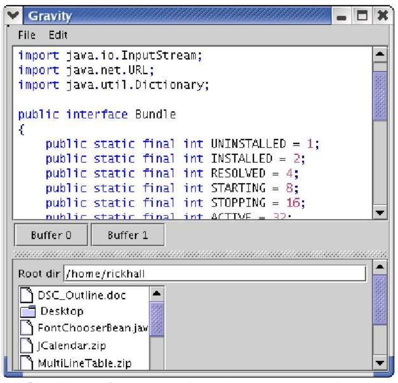
d) Departing building blocks trigger auto-adaptation repairs.



```
import java.io.InputStream;
import java.net.URL;
import java.util.Dictionary;

public interface Bundle
{
    public static final int UNINSTALLED = 1;
    public static final int INSTALLED = 2;
    public static final int RESOLVED = 4;
    public static final int STARTING = 8;
    public static final int STOPPING = 16;
    public static final int ACTIVE = 32;
```

e) Departing building blocks may not be repairable; functionality degrades.



```
import java.io.InputStream;
import java.net.URL;
import java.util.Dictionary;

public interface Bundle
{
    public static final int UNINSTALLED = 1;
    public static final int INSTALLED = 2;
    public static final int RESOLVED = 4;
    public static final int STARTING = 8;
    public static final int STOPPING = 16;
    public static final int ACTIVE = 32;
```

f) Repairs eventually occur when alternative building blocks appear.

Figure 2: Gravity framework prototype usage scenario.

uninstalled components.

The following is an example of the XML representation of an instance descriptor:

```
<bundle>
 <instance
  class="org.office.SpellCheckServiceImpl">
   <property name="Language"
    value="English" type="string"/>
   <provides
    service="org.office.svc.SpellCheckService"/>
   <requires
    service="org.office.svc.DictionaryService"
    filter="(Language=English)"
    cardinality="1..n"
    policy="dynamic"
    bind-method="addDictionary"
    unbind-method="removeDictionary"/>
 </instance>
</bundle>
```

This example describes a component whose implementation is a Java class called `SpellCheckServiceImpl`. An instance of this component has a language property associated with it and implements the `SpellCheckService` service interface. Instances of this component have a dynamic one-to-many dependency on `DictionaryService` service interfaces.

2.3 Framework Usage Scenario

Figure 2 depicts several screen captures of the Gravity framework in action. Snapshots 2a and 2b depict the composition and design environment, which works much like a typical GUI rapid application design tool. Behind the scenes, though, the application's composition connections are being automatically managed and monitored based on component instance meta-data descriptions. The list box along the left side contains the available component types and new component types are automatically discovered as component factories are registered in the underlying service registry. The composed application is live and the user can switch into run-time mode at any time to use it, as shown in figure 2c.

Snapshots 2c, 2d, 2e, and 2f depict a usage scenario where the example application, a simple text editor, undergoes dynamic changes at run time. The text editor in the figure is composed of numerous components. The menu bar, buffer switcher, and file selector components all interact with an “editor service” provided by the editor component. Further, the buffer switcher implements a special “plug-in” service interface, defined in the editor package, so that it can receive events from the editor component in order to update its user interface when buffers are opened and/or closed. The file selector uses a tree renderer service provided by an arbitrary component to render the tree structure of the file system. This composition, depicted in figure 3, is maintained by the Gravity framework.

Figures 2c and 2d depict the transition that occurs when the tree renderer dynamically disappears. Since an alternative tree renderer is available, the framework automatically repairs the file selector's broken dependency. Notice that the new tree renderer uses panels to display the tree structure, instead of a tree widget

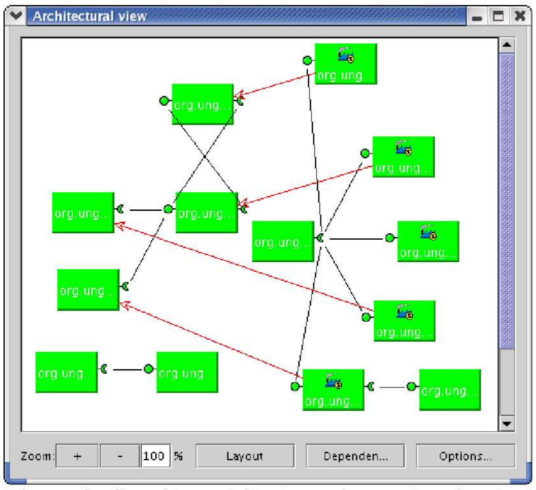


Figure 3: Gravity architecture viewer showing text editor and framework instances.

like the original, but is otherwise functionally equivalent. Figure 2e depicts the result when the second tree renderer dynamically disappears. Since no other alternatives are available, the file selector's dependency is broken and the instance is invalidated. The framework replaces the component with a placeholder (currently an animated "under construction" icon). The final snapshot in figure 2f shows the result of a dynamic arrival of a tree renderer. The framework revalidates and restores a new file selector instance into the application.

3 RELATED WORK

Component models ranging from COM [3] to CCM [9] share concepts with Gravity, but none explicitly support run-time changes other than via programmatic methods. Component-based development environments, such as IBM's Eclipse [11] and WREN [7], are also related to Gravity, but these environments assume that composition cannot occur during application execution.

Service-oriented approaches include OSGi and web services. OSGi defines a framework to deploy services in a centralized environment but leaves service dependency management to programmers. Web services target business application interoperability. Web service composition, as in BPEL4WS [5], is realized through flow models that represent business processes. A web service composition can become the implementation of another service, thus hierarchy is supported. The web service approach, however, does not support automated service dependency management and does not cover deployment.

Composition languages [8], such as CoML [2] 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 towards supporting runtime dynamic changes.

Dynamically reconfigurable systems focus on reconfiguring systems during execution [6]. These systems use explicit architecture models and map changes in these models to the application implementation. An example of such as system is ArchStudio [11] which is a suite that supports run-time reconfiguration of applications for an architectural style called C2. Dynamic reconfigurable systems provide mechanisms to change the structure of a system but do not focus on automatic repairing upon changes.

Recent related research includes autonomic computing, which promotes systems that self-monitor and self-heal, and proactive computing, which promotes systems that try to anticipate user needs to trigger reconfigurations [13]. These computing trends are mostly at a proposal stage and they focus on networked applications, instead of client applications. They do, however, depend on context awareness to some degree and could potentially offer some of the other needed capabilities depicted in figure 1.

4 CONCLUSION

This paper described a research project, called Gravity, that supports a new paradigm for building client-side applications where building blocks may appear or disappear at any time, i.e., they exhibit dynamic availability. Gravity envisions a world where integration decisions are simplified and automated, where context is used to filter which building blocks are part of the current application composition. Gravity supports this vision using a service-oriented component model and framework. The Gravity prototype, described in the paper, was implemented on top of the OSGi services framework. Future work will concentrate on creating predictable applications in the face of ambiguities (such as when multiple candidate services are available), integrating context and semantics, and supporting *plastic* [4] (i.e., fluidly dynamic) user interfaces.

5 REFERENCES

[1] G. Bieber and J. Carpenter. "*Introduction to Service-Oriented Programming*," Whitepaper, Sept. 2001.

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

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

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

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

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

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

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

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

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

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

[12] OTI Inc. "*Eclipse Platform Technical Overview*," 2001.

[13] R. Want, T. Pering and D. Tennenhouse, "*Comparing Autonomic & Proactive Computing*," IBM Systems Journal, July 2002.