A Component Based Infrastructure to Develop Software Supporting Dynamic Unanticipated Evolution

Hyggo Almeida [1], **Angelo Perkusich**[1], **Evandro Costa**[2], **Glauber Ferreira**[1],
Emerson Loureiro [1], **Loreno Oliveira**[1], **Rodrigo Paes**[3]

[1]Embedded Systems and Pervasive Computing Lab
Electrical Engineering and Informatics Center
Federal University of Campina Grande – Campina Grande – PB – Brazil

[2]Institute of Computing – Federal University of Alagoas
Maceió – AL – Brazil

[3]Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro – RJ – Brazil

{hyggo,perkusic,glauber,emerson,loreno}@dee.ufcg.edu.br

Abstract. *This paper presents a component based infrastructure for developing software supporting dynamic unanticipated software evolution. We propose a component model providing mechanisms for managing unpredicted software changes, even at runtime. A Java implementation of the proposed model is also presented. Moreover, a performance evaluation model and an Eclipse-based tool to support composition activities are described. Finally, a pervasive computing middleware application developed using the proposed infrastructure is presented.*

Resumo. *Neste artigo apresenta-se uma infra-estrutura baseada em componentes para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. Propõe-se um modelo de componentes que provê mecanismos para gerenciar alterações no software não previstas em projeto, inclusive em tempo de execução. Apresenta-se uma implementação em Java do modelo de componentes e um modelo analítico para avaliação de desempenho. Para dar suporte às atividades de desenvolvimento, propõe-se uma ferramenta baseada na plataforma Eclipse. Por fim, apresenta-se uma aplicação da infra-estrutura proposta para o desenvolvimento de um middleware para computação pervasiva.*

1. Introduction

Various studies have pointed evolution as responsible for up to 90% of the total cost of software development [Ebraert et al. 2005]. The impact of evolution on existing design and code is more significant when software requirement changes have not been anticipated. Unanticipated software evolution is a key issue on software engineering, since it is strongly related to software development and maintenance cost and time.

By definition, "unanticipated software evolution is not something for which we can prepare during the design of a software system" [1]. This way, any support for unanticipated software evolution must not require developers to specify which part of the software could evolve. Any part of the software must inherently support evolution, and developers should not bother themselves about the mechanisms that allow this evolution. Moreover, in the case of systems with frequent requirement changes or when the execution of such systems cannot be interrupted, evolution must be managed at runtime.

This seamless evolution support is not provided by existing techniques of Software Engineering, such as application frameworks [Fayad et al. 2000], component based systems [Crnkovic 2001], service oriented architectures [Papazoglou 2003], and plug-in based development [Mayer et al. 2003]. Although some of these approaches provide flexibility

---

[1]FUSE Workshop - http://www.informatik.uni-bonn.de/~gk/use/fuse2004/

for changes even at runtime, they have not been conceived to support unanticipated changes. Developers have to point out potential parts to be changed in the future through extension points, services interfaces, framework hot spots, plug-in interfaces, etc.

The major problem arises when a part of the software which was thought to be fixed has to change. Unanticipated changes force developers to extensively modify existing software architecture, design, and code. A change is considered "unanticipated" when its implementation does not depend on hooks encoded in previous versions of the changed software [Kniesel et al. 2002].

In this paper we introduce a component based infrastructure to develop software supporting dynamic unanticipated evolution. We propose a component model, named COMPOR COMPONENT MODEL SPECIFICATION (CMS), which allows changing any part of the software, by removing and/or adding components, even at runtime. CMS promotes dynamic unanticipated software evolution through a simple and lightweight design model. Component based concepts, such as containers and components, are used to build applications based on a hierarchical composition. We present a Java implementation of the CMS, called JAVA COMPONENT FRAMEWORK (JCF). Due to the simplicity of the object oriented framework model, it can be implemented using other programming languages. An implementation based on Python and another one in C++ are discussed in the concluding remarks of this paper. By using JCF, it is possible to develop Java applications that can be changed during runtime, even for unpredicted changes. A performance evaluation model for CMS-based architectures is also described. This performance model is very useful to define which CMS architecture is more suitable for a specific application. Moreover, we present a set of Eclipse-based tools (`http://www.eclipse.org`) for supporting the software composition activities, called COMPONENT COMPOSITION TOOLS (CCT). Finally, a case study for the proposed infrastructure is described: a pervasive computing middleware.

The remainder of this paper is organized as follows. Section 2 describes the CMS. Section 3 introduces the JCF. In Section 4 we describe a performance evaluation model for the CMS. Section 5 introduces the CCT. The pervasive computing middleware developed using CMS/JCF is presented in Section 6. Related works are discussed in Section 7. Finally, concluding remarks are presented in Section 8.

2. Component Model Specification

According to the COMPOR COMPONENT MODEL SPECIFICATION (CMS), a component based system is described as a composition of two kinds of entities: functional components and containers. Functional components are software entities implementing application-specific functionalities, making them available by means of services and events. A functional component is not composed of other components, that is, it has no child components. Functional components represent the atomic architectural elements of the software application according to the CMS.

Containers are software entities that implement no application-specific functionalities. A container controls the access to the services and events provided by its child components, which may be functional components or other containers. Functional components are made available by inserting them into containers. It is necessary to have at least one container, named *root container*, in order to insert functional components into it, and then run the application. Each container has tables of services provided by and events of interest to its child components.

2.1. Component Deployment Model

After inserting a new component into a given container, the tables of services and events for each container up to the root of the hierarchy must be updated accordingly. This is necessary in order to make available the services provided by the inserted component as well as to allow the notification of the occurrence of events that are of interest to the component. Figure 1 depicts the component deployment process and its steps are detailed as follows.
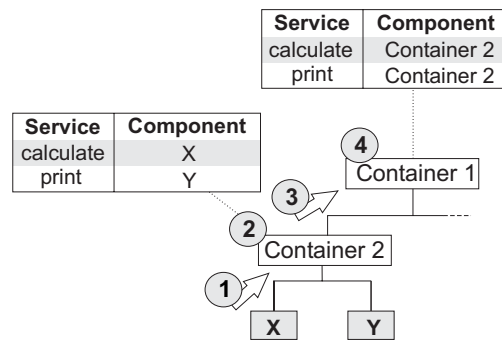
Figure 1. Component deployment - updating service and event tables.

1. A component *X* implementing the "calculate" service is added to *Container 2*.

2. *Container 2* updates the service table adding the services provided by the component *X* (the same occurs for the event table).

3. *Container 2* asks its parent container, *Container 1* in this case, to update its service table, by adding the services provided by *Container 2*.

4. *Container 1* updates its service table.

After the execution of these steps, the services provided by the component *X* can be accessed from any component in the hierarchy without an explicit reference to it. Note that a component has only reference to its parent container. In the case of two or more components having services or events with the same identifier, an *alias* is used. Thus, it is possible to register a nickname for each service, allowing services providers to coexist within the same application and be diversified in terms of non-functional features. The remove operation is similar to the deployment, but after removing a component the next invocations for its services will not work.

2.2. Interaction Model

The interaction model is based on services and events. In the first case any component may invoke a service of another component, even when the component belongs to other container. The interaction based on events focuses on the announcement of a state change in a given component to all the interested components. In both cases there is no explicit reference among components.

2.2.1. Service based interaction

As said before, after inserting a component into a given container, its services are made available to any other component in the application. Therefore, assuming the existence of a service named "save" implemented by a component *K* of the application, the execution of this service can be requested by a component *X*, without an explicit reference to *K*. In Figure 2 such an interaction process is illustrated and detailed as follows.

1. Component *X* requests the execution of the "save" service to its parent container.

2. Based on its service table, *Container 2* verifies that no child component implements the "save" service.

3. *Container 2* forwards the request to its parent container, in this case *Container 1*.
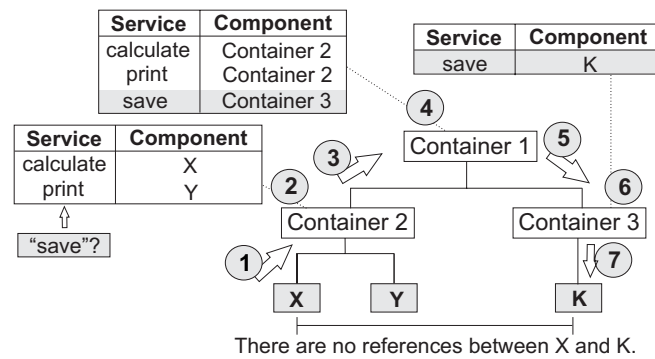
147

Figure 2. Service based interaction - No explicit references.

4. *Container 1*, according to its service table, verifies that one of its children implements the "save" service, *Container 3* in this case. The *Container 1* sees *Container 3* as the component that implements the requested service.

5. *Container 1* then forwards the service request to the *Container 3*.

6. *Container 3* does not implement the service but has a reference to the component that implements the requested service, and forwards the request to it, in this case component *K*.

7. Component *K* executes the "save" service and returns the result following the reverse path, back to the requester.

It is important to point out that there are no references between the component requesting the service (*X*) and the component that provides it (*K*). Thus, it is possible to change the component that provides the "save" service without modifying the rest of the structure.

2.2.2. Event based interaction

When an event is announced by a given functional component, all the components in the hierarchy of the application that are interested in the event must be notified. The interaction based on events is also controlled by containers, and thus there are no direct references among functional components. This process is shown in Figure 3 and the steps are detailed as follows.

1. The component *X* announces an event named "Event A".

2. The announcement is directly received by its parent container (*Container 2*), which verifies if any of its child components have to be notified about the event, by inspecting the event table.

3. *Container 2* forwards the event to the interested components, in this case only the component *Y*.

4. *Container 2* then forwards the event to its parent container (*Container 1*).

5. *Container 1*, according to its event table, forwards the event to those interested on it, except the one that announced the event (*Container 2*). Since *Container 1* is the root of the hierarchy, there is no parent container to forward the event. Thus, the event is only forwarded to *Container 3*.
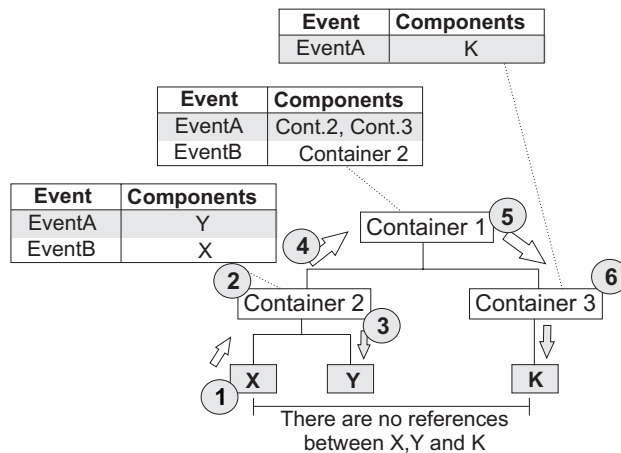
Figure 3. Event based interaction - event notification without explicit references.

6. *Container 3* forwards the event according to its event table that in this case is component *K*.

As can be seen in Figure 3, there are no references between the component that announced the event (*X*) and those interested on it (*Y* and *K*). Thus, the component that announces the event can be changed without modifying the rest of the structure.

2.3. Overriding Services: the Black-Box Inheritance Mechanism

Due to the container-mediated deployment and interaction models, CMS provides mechanisms to override services via "black-box inheritance". In other words, it is possible to reuse component services without extending the component code, even at runtime. This is possible because a component can require a service implemented by itself. Since the services provided by functional components are accessed via the container, it is only necessary to publish internal component functionalities as external services and access them via a container. Figure 4.a illustrates an example of this process. Consider the "readFile" service, which is implemented by the sequence of functionalities: "buffering" and "IOoperation". If the internal functionalities are published as services, *Component1* can access them via container. Then, the "readFile" service is decoupled from "buffering" and "IOoperation" functionalities.
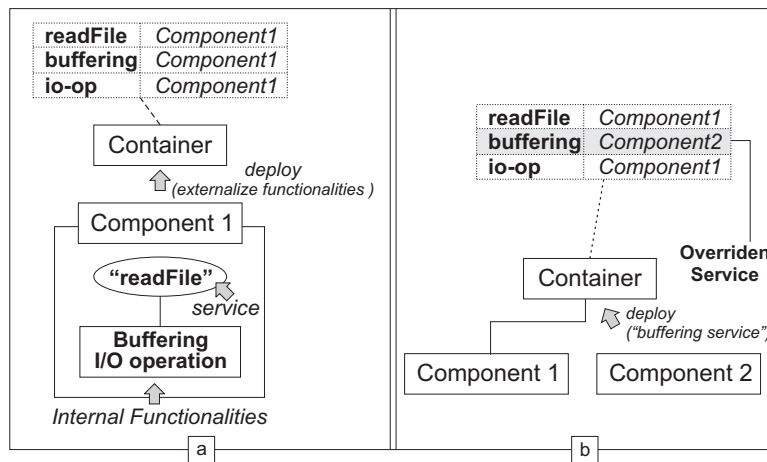

Figure 4. Service overriding - internal functionalities as component services.

To override the "buffering" internal functionality, for example, it is only necessary to deploy a component that provides a service with the same name. Figure 4.b presents

the service overriding process. In this figure, *Component2* overrides the service "buffering" of the *Component1*. Since there are no references to the *Component1*, this process, which is based on the *Template Method* and *Strategy* design patterns [Gamma et al. 1995], can be performed at runtime. After deploying the *Component2*, the "readFile" service of *Component1* becomes based on the "buffering" service implemented by the *Component2*. It is important to note that *Component2* does not have to extend *Component1*, it is only necessary to know the provided and required services.

2.4. Recursive Composition: Applications as Components

One can think that a flat architecture could be always better. It could be more interesting in terms of performance and will not require the usage of several containers. Also, it will work similarly to service oriented architectures, like Jini [Waldo 1999]. However, the main motivation for a hierarchy of containers is that it allows to maintain cohesion of functionalities provided by their child components. It makes possible to reuse entire containers without needing to understand the internal components or other containers, at any level of the hierarchy. Also, it allows recursive composition of applications, since root containers can be viewed as components for other containers. Therefore, an application can be built by integrating containers of various CMS based applications (Figure 5).
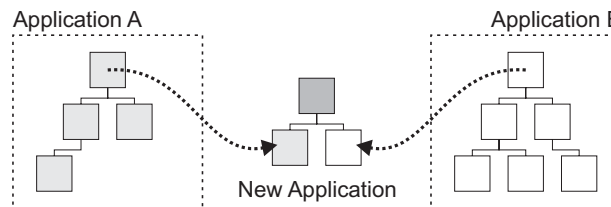

Figure 5. Composition of applications.

Based on the service and event models, black-box inheritance and recursive composition, the CMS supports all kinds of evolution scenarios: component change, addition, and removal; service and event changes; and architectural changes. In face of unanticipated evolution, an entire application could be dynamically changed. It is only necessary: i) to identify which components will change; ii) to identify which are their dependencies (services and events) and, if necessary, change them; iii) and finally change the identified components. The effort needed to perform this evolution depends on the functional cohesion and complexity of the application. In fact, it could be hard but using the CMS it will be possible.

3. Java Component Framework

The JAVA COMPONENT FRAMEWORK (JCF) is a Java implementation of the CMS. The JCF design is based on the *Composite* design pattern [Gamma et al. 1995], which can be applied to hierarchical architectures. Figure 6 shows a simplified version of the JCF class diagram, describing its main methods. `Container` and `FunctionalComponent` classes are instantiated for containers and functional components, respectively. The abstract class `AbstractComponent` assures the recursive composition [Gamma et al. 1995]. Thus, containers are not aware if their children are functional components or other containers. Additionally, it implements the accessor methods. The methods declared in the class `AbstractComponent` are differently implemented by `FunctionalComponent` and `Container` classes, for both the service and the event interaction models.

The service interaction model is implemented through iterative invocations of the `doIt` and `receiveRequest` methods. Such methods are invoked by the components and containers of the hierarchy until the service provider component is located. The function of the `doIt` method is to forward the service request, in a bottom-up way, until reaching the container that contains the reference to the provider. When this occurs, the `receiveRequest` method is invoked, in a top-down way, until reaching the functional
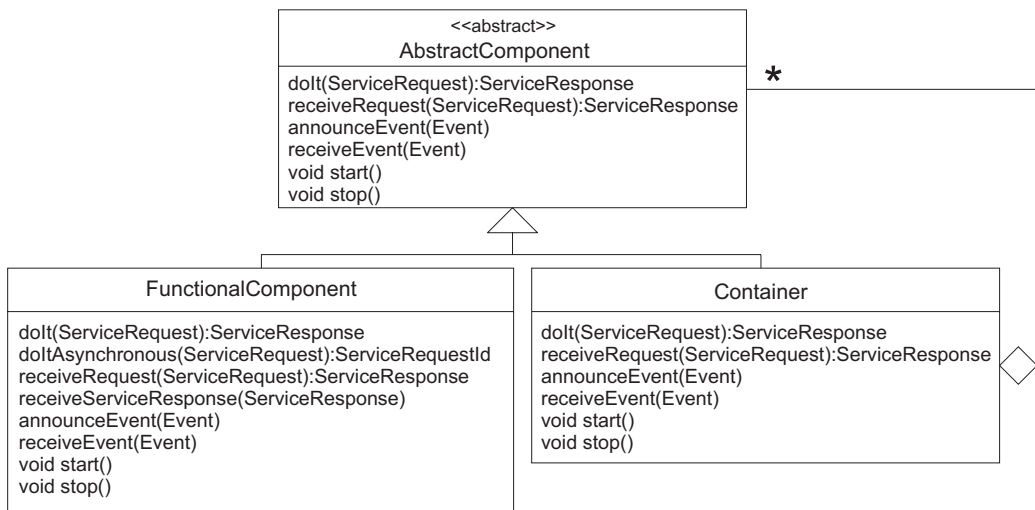
Figure 6. Simplified JCF class diagram.

component that implements the service (Figure 7). The syntax for the service invocation methods are `doIt(ServiceRequest)` and `receiveRequest(ServiceRequest)`, where `ServiceRequest` is an object that encapsulates a service name and the parameters needed to execute the service. The result of those methods is a `ServiceResponse`, which encapsulates the service execution result or the exception, if it occurs.
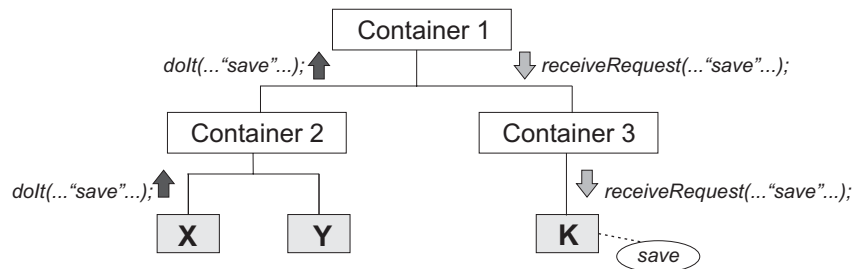

Figure 7. Execution of the `doIt` and `receiveRequest` methods.

The JCF also implements an asynchronous version of the service based model. The asynchronous interaction implementation is based on the *ActiveObject* design pattern [Vlissides et al. 1996]. A component asynchronously invokes a service through the method `doItAsynchronous` and receives a request identifier (`ServiceRequestId`). Then, a new thread is started to request the service through the method `doIt`. When the return from `doIt` occurs, it invokes the method `receive ServiceResponse` for the service requester component, forwarding the service request and the service identifier. Based on the service identifier, the requester component identifies to which invocation the reply refers to.

The implementation of the event based interaction model is based on the *Observer* [Gamma et al. 1995] and *ActiveObject* [Vlissides et al. 1996] design patterns. The functionality is implemented through the asynchronous invocation of the method `announceEvent` to announce events to the parent containers (bottom-up). On the other hand, the invocation of the method `receiveEvent` notifies the events to the interested components (top-down), as occurs with services.

Besides the interaction models specified by the CMS, the JCF implements initialization properties for components. Moreover, JCF provides a mechanism for starting and stopping the execution of the components. The initialization properties are stored in a table for each functional component and can be accessed through the `getInitialization`

`Parameter(String)` method, whose argument is the name of the required initialization parameter. The component initialization is implemented by the `start` and `stop` methods. For containers, these methods start/stop all of its components through the invocation of their respective `start` and `stop` methods. For functional components, these methods are template methods [Gamma et al. 1995] that invoke abstract methods implemented by the component developer. These methods initialize/interrupt the execution according to the component needs.

3.1. Security Support

As described in Section 2.1, an alias is used to uniquely identify services and events with the same name for different components. However, such a strategy introduces a security problem into the model. For example, it is possible to interpose a provider *X* between another provider *Y* and its clients in order to intercept the client requests towards *Y*. This may represent an intrusive way to make something undesirable in the system, since the interposed provider *X* may be seen as an intruder.

As this security issue is not tackled by the component model, the JCF must provides means for dealing with security policies for the interaction and deployment models. Such policies must then be satisfied when some service is requested or an event is announced as well as a component is inserted or removed from a container. This security infrastructure, shown in Figure 8, was developed using aspect oriented programming, with AspectJ [Kiczales et al. 2001]. Aspects have allowed to hide the complexity of the security mechanism from the developer as well as to simplify the development of systems without security requirements. The security mechanism illustrated in Figure 8 is explained as follows.
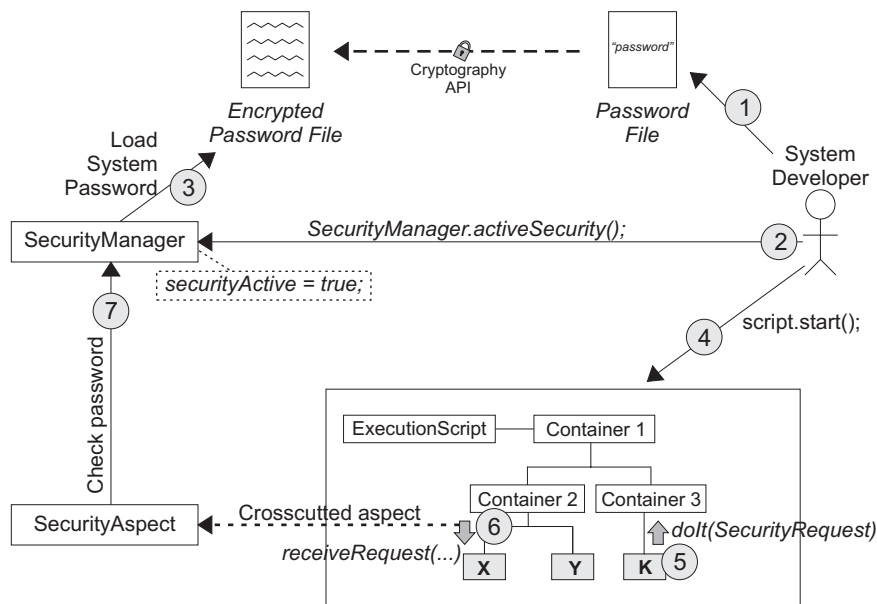

Figure 8. Aspect oriented security architecture.

1. The application developer creates a ".security" file containing the password for accessing the system as well as the service access policies. Then, uses the Java cryptography API to encrypt the file.

2. When developing the application, the security mechanism should be activated calling the `activeSecurity()` method of the `SecurityManager` singleton class. This operation defines that all service invocations, event announcements and component additions must be verified.

3. The `SecurityManager` retrieves the password and the policy information and stores them in memory.

4. After starting the root container, all of its components are also started and the application runs by means of a sequence of service invocations and event announcements.

5. A component invokes a service. With the security activated, the service requester component must forward a `SecurityServiceRequest` instance as parameter, containing the system password.

6. The component receives the request via the `receive Request` method, then the `SecurityAspect` aspect intercepts the method invocation and asks the `Security Manager` to verify the request password.

7. `SecurityManager` verifies the request password and allows the service execution. Otherwise, a `ComporSecurity Exception` is thrown.

4. Performance Analysis

There is a trade-off between flexibility and performance in the CMS. Although it allows dynamic composition of components, both its deployment and interaction models introduce an impact on the software performance when requesting services and announcing events. Considering performance a critical non functional requirement for some application domains, we propose a performance evaluation model that allows evaluating the performance of an application with specific architectures based on the CMS. Through this evaluation model, it is possible to identify and reduce potential overheads caused by the architectural design.

There are three main operations defined in the CMS to be evaluated: component deployment, service request, and event announcement. As mentioned before, the architecture of an application according to the CMS can be represented by a tree. Thus, the performance evaluation for these operations is based on a tree structure. The performance is measured based on the time spent to perform the two main operations for implementing the CMS: method invocations and accesses to data structures. Each tree edge represents a method invocation operation, followed by the invocation of a set of related methods, and accesses to data structures that store the event and service tables. To exemplify the application of the evaluation model, the mean time of method invocations and access to Java hash tables are considered.

4.1. Component Deployment Analysis

The component deployment operation is concerned with the insertion of functional components into containers, making their services and events available to other components. As mentioned before, after the insertion of a component, the tables of services and events for each container up to the root of the hierarchy is updated accordingly. Therefore, the deployment operation time can be evaluated as: $T_d = t_r + d \times (t_s + t_e)$, where: t_r is the mean time to register the new component on its parent container; d is the depth of the new component (the depth of a node is the length of the path from the root to that node); t_s is the mean time to register the provided/required services to the new component on all the containers between its parent and the root; and t_e has the same meaning as t_s, but announced/interested events are registered instead.

For instance, Figure 9 presents the deployment of the component n, which results in registering the component to its parent, and registering its provided/required services and announced/interested events to two containers. Thus, supposing $t_r = 20\mu s$ and $t_s = t_e = 22\mu s$, the time of the operation using the JCF is estimated as $T_d = 20\mu s + 2 \times (22\mu s + 22\mu s) = 108\mu s$.
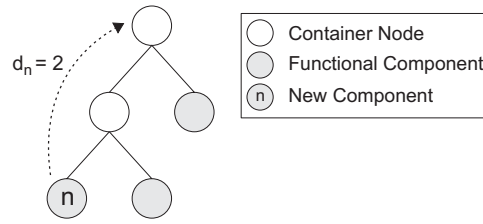
Figure 9. Component deployment performance evaluation.

4.2. Service Request Analysis

According to the CMS, when a service request is performed, the request is propagated through the hierarchy reaching the service provider, if it exists. Thus, the time for a service request operation can be given by $T_s = t_n + (d_r + d_p + 1) \times t_m$, where: t_n is the mean time to create a new service request; d_r is the depth of the node of the requester component; d_p is the depth of the node of the provider component; the constant $(+1)$ refers to an extra query, performed by the requester component to itself (for the scenarios where, possibly, the own component implements some desired service); and t_m is the mean time to access the data structures. Figure 10 illustrates the computation time of a service request in the JCF. In this case, the component *r* requests a service provided by the component *p*. Supposing $t_n = 25\mu s$ and $t_m = 30\mu s$, the time for this operation is given by $T_s = 25\mu s + (3 + 2 + 1) \times 30\mu s = 205\mu s$.
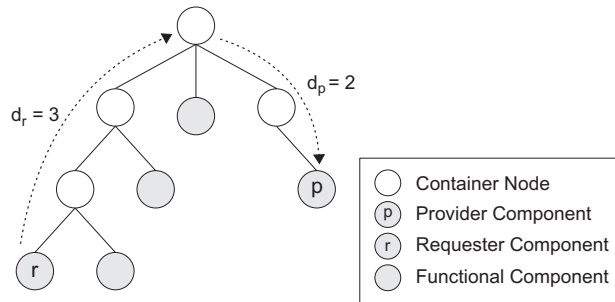

Figure 10. Service request performance evaluation.

4.3. Event Announcement Analysis

Unlike the service request operation, when an event is announced, the announcement is propagated through the hierarchy until reaching all the interested components. Since several components can be interested in the same event, various event targets may exist. If the formula for service request evaluation is used, an edge could be counted twice or more. In order to deal with this problem, besides containers and functional components, the concept of *common container node* is defined.

A *common container node* is a container node that: (i) contains two or more child nodes; (ii) it is composed of at least two "k nodes" (interested, announcer, or another common node), where each "k node" belongs to different subtrees. Furthermore, even if there are no interested components, the event is propagated to the root node. In this way, every root node is also defined as a *common container node*. The time for an event announcement operation is defined as: $T_e = t_a + t_t + \sum_{i=1}^{n} g_{k_i} \times t_c$, where: t_a is the mean time to create a new event announcement; t_t is the mean time to create and fire a new thread to deliver the event to all the interested components; g_{k_i} is the number of edges from each "k" to its *common container node*; and t_c is the mean time to query data structures about event interests.

For example, consider the tree representing an architecture of an application shown in Figure 11. In this figure, there is one component that announces an event and six components that are interested in it. According to this hierarchy and supposing $t_a = 20\mu s$,

$t_t = 80\mu s$ and $t_c = 30\mu s$, the time of the operation is $T_e = 20\mu s + 80\mu s + \sum_{i=1}^{11} g_{k_i} \times 30\mu s = 550\mu s$.
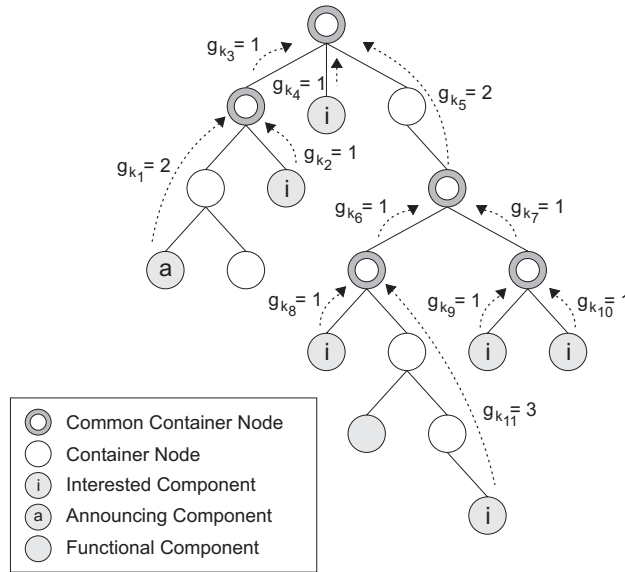

Figure 11. Event announcement performance evaluation.

4.4. JCF Code Profiling

In order to validate the evaluation model and obtain real performance evaluation measures we have performed a code profiling on the JCF, making possible to compare real data with the analytical evaluation results. The JProfiler (`http://www.ej-technologies.com`) profiler was used to verify the real cost for component deployment and execution of events and services. A dedicated Pentium IV 2.8 GHz machine, with 512MB of main memory, running Sun Java Virtual Machine version 1.4.2 and Windows XP Service Pack 2 were used to execute code profiling. Figure 12 presents the comparison between the results of the performance evaluation model and the profiling results. The x-axis should be read as: *depth of new nodes*, for the deployment of components; *total path length*, for the service requests; and *number of edges*, for the event announcement.
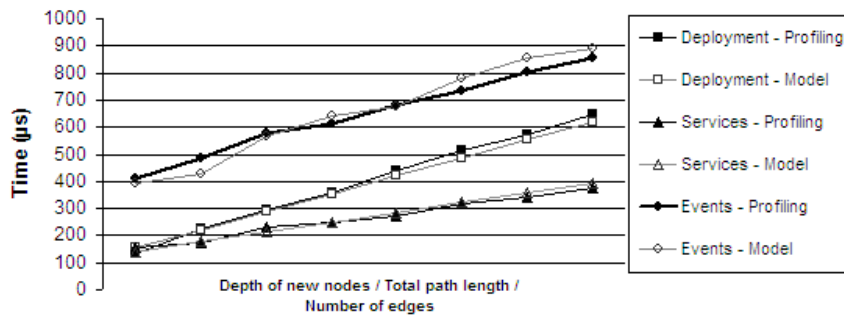

Figure 12. Model results vs. Profiling results.

For our experiments we have performed some changes in the component structures presented in the sections 4.1, 4.2, and 4.3. To evaluate the performance of component deployment, we have changed the tree depth. For the service requests, we have changed the tree depths for the requester and provider and explored different configuration of components for each combination of provider and requester depths. Finally, for event announcements, we have changed the number of interested components, exploring different configurations of components for each number of interested components.

4.5. Discussion of the Evaluation Results

The result of the evaluation indicates that the performance impact can be minimized by managing the depth of the container hierarchy. For example, consider an architecture where there is only one container (the root) and all its children are functional components, i.e, their depth is one. In this case: a component deployment operation is performed through only one service/event registering operation; a service request is performed via three accesses to the data structures; and an event announcement is performed through $(number\ of\ interested\ components + 1)$ accesses to data structures, with only one extra method execution compared to an usual *Observer* pattern implementation [Gamma et al. 1995].

Although improving the performance of the operations, a flat hierarchy reduces modularity, cohesion, and flexibility of the software architecture. It occurs because the use of containers allows the modularization of related components, making possible to change entire containers to other containers or components, as described in Section 2.4.

On the other hand, the deeper a hierarchy is, the higher is the number of method executions and accesses to data structures, and consequently the performance is reduced. An example of this kind of "deep hierarchy" is illustrated in Figure 11, where fifteen accesses to data structures occur for announcing an event to six interested components.

Based on the analysis of profiling results, we conclude that the measures obtained are very close to the analytical results, with mean errors around 5% for all operations. Probably, these errors are related to potential optimization operations performed by the Java Virtual Machine. Therefore, depending on the performance and flexibility requirements of an application, either a deeper or flatter hierarchy is more suitable. The evaluation model is useful for evaluating the performance of specific architectures still at design time.

5. Component Composition Tools

The `Component Composition Tools` (CCT) is a set of Eclipse plug-ins to develop software supporting dynamic unanticipated evolution based on the CMS. The main motivation for building the CCT was the significant effort required to compose a system using the JCF. For large scale applications, the programming effort to build several components, to compose various containers, and to define many services and events can be very high without automation tools.

The CCT main tools have been built over the Eclipse platform core: *component pallete/manager*, to deploy and make components available to be reused; *component tree/inspector*, to manage and configure the application components; *component test*, to perform integration tests; and a *component description wizard*, to describe new components. Moreover, a *component editor* is being constructed over the *AspectJ Development Tools* (`http://www.eclipse.org/ajdt`) to ease the component development activities. The support for aspects provided by the CMS is not covered in this paper.
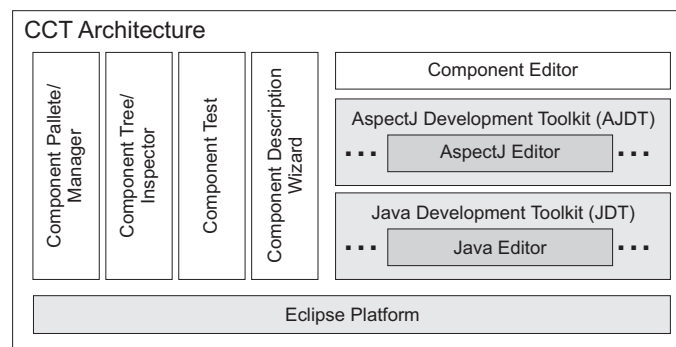

Figure 13. Architecture of the CCT.

The graphical interface of the CCT is implemented as a set of Eclipse views and wizards. A CCT perspective and project nature were created to provide a customized composition workspace for the developer. Figure 14 depicts the main screen of the CCT perspective.
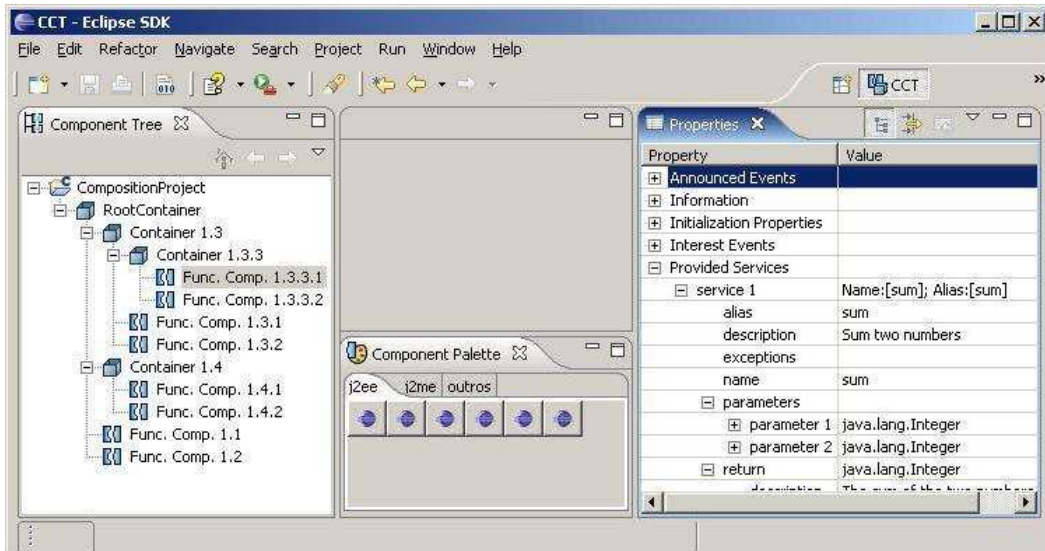

Figure 14. CCT perspective on the Eclipse platform.

With CCT, it is possible to compose CMS applications by assembling pre-existing components. However, it does not provide support for running and monitoring applications. For that, we are working on developing a Component Application Server (CAS) that runs on top of Oscar – an implementation of the *Open Services Gateway Initiative* (OSGi) (`http://www.osgi.org`). CAS provides an execution environment for the deployment and execution of CMS/JCF components and applications. Also, it manages component life cycle and controls component versioning. Oscar/OSGi is used for managing functionalities related to dynamic class loading. Using CCT and CAS, developers have full support for developing, composing and running applications based on CMS.

6. Case Study: Wings Pervasive Middleware

Wings is a middleware for pervasive computing that is guided by three issues: context-sensitivity; networking support flexibility; and interoperability both in terms of networking protocol stack and programming language [Loureiro et al. 2005]. The basis of the middleware lies on the concepts of *resource*, *context* and *peer*. We define a *resource* as an entity with a description, through which it can be shared, discovered and downloaded, such as an audio file. A *context* encapsulates information about a local peer and the environment in which it is immersed. Finally, *peers* are defined as network nodes having the following set of capabilities: search and sense other peers; share, discover, and download resources, as well as deliver context information.

Due to the sensing capability, Wings has been designed for "infrastructureless" environments. Therefore, the communication between peers must be performed in an ad-hoc way. This characteristic enhances the applicability of Wings in the world of pervasive computing, where the infrastructure is something we cannot always count on. This approach provides the necessary tools to develop applications (*Winglets*) for ad-hoc like pervasive environments such as mobile virtual communities and mobile file sharing. Using Wings, pervasive applications could take advantage of multiple configurations by performing host discovery over different network infrastructures, possibly at the same time. Based on this approach, an application could, for example, discover hosts

through *UPnP* (http://www.upnp.org), *JXTA* (http://www.jxta.org) and *Zero-conf* (http://www.zeroconf.org) protocols. This improves the acquisition of context information, since more hosts can be discovered by the applications.

However, it is very difficult to predict which of such protocols will supply the needs of different applications. Moreover, mobile devices are still very limited concerning memory and storage capacities. Therefore, it would not be reasonable to embed in such devices all the existing network infrastructure protocols for each of their wireless interfaces. It becomes necessary a mechanism for inserting and removing such implementations from a device, whenever needed. For that, we use the CMS to encapsulate the peer discovery algorithm and context information mechanisms in software components. Such components, namely Network Infrastructure Components (NICs), may be plugged in and out from the middleware even at runtime (Figure 15).
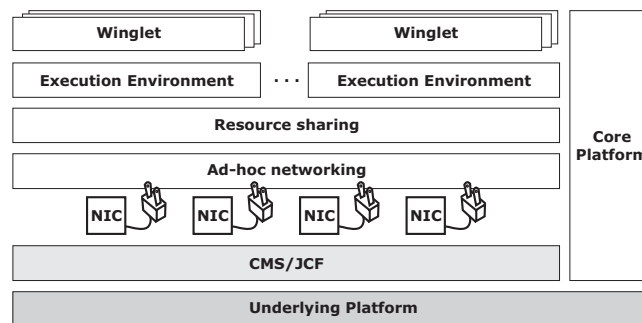

Figure 15. Wings Architecture.

The CMS successfully provides an effective way of changing *NICs* at runtime, without affecting the rest of the middleware. This is an important feature in a pervasive environment, where the networking protocols involved may change, but users do not want to stop their tasks for replacing one protocol for another. In other words, this process should be performed transparently.

7. Related Work

Different kinds of component models have been proposed. Some examples are *Sun JavaBeans* and *Enterprise Java Beans* (http://java.sun.com), and the *CORBA Component Model* (http://www.omg.org). Such models have been successfully applied for constructing corporative applications and their middleware implementations provide many interesting services for enterprise software development. However, these models were not conceived to support dynamic unanticipated software evolution. In some cases, their middlewares provide mechanisms and services to perform dynamic changes, but this is not defined in a component model level. The lack of this feature makes difficult the construction of systems supporting unanticipated evolution.

CMS has also some similarities with service oriented architectures: service publishing and provision, transparency of the service provider, flexibility for changes, among others. For instance, Jini [Waldo 1999] is a Java-based technology for the provisioning of services among network nodes. In Jini, services are advertised and discovered in central repositories, like a distributed CMS single (root) container. The work presented in [Handorean and Roman 2003] describes a Java middleware for providing services in ad-hoc networks. Such a middleware is based on a distributed service registry, where each node of the network is able to provide and use services. Other example is the OpenWings (http://www.openwings.org) framework, which aims at providing service provisioning features targeted to dynamic networks. In these works, dynamic evolution is not provided. In the work of Piccinelli et al [Piccinelli et al. 2003] the main focus is the dynamic composition of services, and recursive composition of services is also allowed. However, dynamic features are only related to service loading, unanticipated changes are not tackled.

In the context of dynamic composition models, we highlight HADAS [Ben-Shaul et al. 2001]. In HADAS, the focus is the interoperability between distributed components. The reflection concept is used to define the dynamic composition. The developer should define a set of *fixed* behaviors and another set of *extensible* behaviors for the application. Only *extensible* behaviors can be dynamically composed and changed. Considering that the changes cannot be predicted, the definition of fixed components imposes many difficulties and in some cases makes no sense, since any component can be eventually changed. Another work is Gravity [Cervantes and Hall 2004] which puts concepts from service and component orientation together for defining a model that supports the adding and removal of components at runtime. For these works, there are no mechanisms to provide recursive composition and the dynamic evolution is only allowed for explicitly defined non fixed parts.

8. Concluding Remarks

This paper presented a component based infrastructure to develop software supporting dynamic unanticipated evolution. We introduced a model, named CMS, which provides mechanisms for managing dynamic changes in the software on the fly, even if they have not been anticipated. A Java implementation of the CMS that allows developing Java applications supporting dynamic unanticipated software evolution was also presented. To make possible a large scale development, an Eclipse-based tool to support the composition activities, called CCT, was introduced.

Moreover, a performance evaluation model was described. Based on the model, the designer can identify CMS-based architecture that is more suitable for a specific application, taking into account performance and flexibility issues, still at design time. Also, we described the implementation of a complex middleware for pervasive computing, which uses the CMS as the key technology to provide flexibility.

The infrastructure presented in this paper represents a novel engineering support for constructing applications supporting to unanticipated evolution. Using CMS, JCF, and CCT, applications can be developed based on reuse, besides improving flexibility and reducing maintenance and evolution time and costs. In what follows, some current efforts and future perspectives are discussed.

Multi-Language Implementations

Multi-language implementations are very important to consolidate the CMS , and also to apply it to different contexts and platforms. For that, besides the Java implementation of the CMS introduced in this paper, a `Python Component Framework` (PCF) and a `C++ Component Framework` (CCF) are being developed. The simplicity of the CMS makes possible to implement dynamic composition without requiring object oriented languages complex mechanisms. For Python, which supports threads, dynamic loading, and dynamic binding as native features, the implementation is even more straightforward. In the case of C++, we had some problems to integrate these features from different sources. They are being applied to different contexts: the PCF is useful for rapid prototyping and the CCF is being applied to Linux-based embedded systems. We are developing the PCF and CCF as close as possible to the JCF design, to provide the same dynamic software composition infrastructure for the developer, regardless of the language.

Future Trends

The main feature trends related to the research reported in this paper are the support for non functional requirements and for formal dependency verification. The former is related to the description and implementation of non functional requirements of the components. Operation time, performance, and real-time restrictions, among other information, must be present in the component description available to the developers. For some domains, it is very important to define if two components could be interchanged, for example. The last is concerned with the formal description of the component interface - services and events.

This allows verifying if the dependencies (required services and events) of all components are being provided according to the formal specification. Also, it will make possible to determine if a remove or change operation could be performed while still maintaining the specification of the components and the system correct.

References

Ben-Shaul, I., Holder, O., and Lavva, B. (2001). Dynamic Adaptation and Deployment of Distributed Components In Hadas. *IEEE Trans. Softw. Eng.*, 27(9):769–787.

Cervantes, H. and Hall, R. S. (2004). Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 614–623. IEEE Computer Society.

Crnkovic, I. (2001). Component-based Software Engineering - New Challenges in Software Development. In *Software Focus*, volume 4, pages 127–133. Wiley.

Ebraert, P., Vandewoude, Y., D'Hondt, T., and Berbers, Y. (2005). Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'05)*.

Fayad, M., Johnson, R., and Schmidt, D. (2000). *Building Application Frameworks*. Wiley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Handorean, R. and Roman, G.-C. (2003). Secure Service Provision in Ad Hoc Networks. In *Proc. of the First International Conference on Service Oriented Computing*, volume 2910 of *Lecture Notes in Computer Science*, pages 367–383, Trento, Italy. Springer Verlag.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.

Kniesel, G., Noppen, J., Mens, T., and Buckley, J. (2002). 1st Int. Workshop on Unanticipated Software Evolution. In *ECOOP Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag.

Loureiro, E., Oliveira, L., Almeida, H., Ferreira, G., and Perkusich, A. (2005). Improving flexibility on host discovery for pervasive computing middlewares. In *3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble, France. ACM Press.

Mayer, J., Melzer, I., and Schweiggert, F. (2003). Lightweight Plug-In-Based Application Development. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 87–102. Springer-Verlag.

Papazoglou, M. P. (2003). Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proc. of Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Rome, Italy. IEEE.

Piccinelli, G., Zirpins, C., and Lamersdorf, W. (2003). The FRESCO Framework: An Overview. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops*, pages 120–123, Orlando, USA. IEEE Computer Society.

Vlissides, J., Coplien, J., and Kerth, N. (1996). *Pattern Languages of Program Design 2*. Addison-Wesley.

Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82.