An Architecture to Support Dynamic Composition of Service Components

David Mennie, Bernard Pagurek

Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, Canada, K1S 5B6
{dmennie, bernie}@sce.carleton.ca

Abstract

The creation of composite services from service components at runtime can be achieved using several different techniques. In the first approach, two or more components collaborate while each component remains distinct, and potentially distributed, within a network. To facilitate this, a new common interface must be constructed at runtime which allows other services to interact with this set of collaborating service components as if it was a single service. The construction of this interface can be realized with the support of a service composition architecture. In the second approach, a new composite service is formed where all of the functionality of that service is contained in a single new component. This new service must be a valid service, capable of the basic set of operations that all other services can carry out. Our goal is to design an architecture to support the runtime creation of composite services, within a specific service domain, using existing technologies and without the need for a complex compositional language. We make extensive use of a modified Jini infrastructure to overcome many of the shortcomings of the JavaBeans component model. In this paper, we compare techniques for dynamic service composition and discuss the requirements of an infrastructure that would be needed to support these approaches. We also provide some insight into the types of applications that would be enabled by this architecture.

Keywords: Dynamic service composition, runtime component assembly, component-based services

1 Introduction

One of the goals of component-oriented programming has traditionally been to facilitate the break up of cumbersome and often difficult to maintain applications into sets of smaller, more manageable components [7]. This can be done either statically at design-time or load-time, or dynamically at runtime. Selecting ready-made components to construct an application is sufficient for a relatively straightforward system with specific operations that are not likely to change frequently. However, if the system has a loosely defined set of operations to carry out, components must be able to be upgraded dynamically or composed at runtime. It is this need for dynamic software composition that we will examine in this paper.

2 Defining the Problem

Our research group has previously approached the problem of dynamic software composition as it relates to high-availability systems. Before we define the approach to runtime composition taken in this paper, we will briefly describe our previous experiences in the area of software hot-swapping.

2.1 Software Hot-swapping

Software hot-swapping is defined as the process of upgrading software components at runtime in systems which cannot be brought down easily, cannot be switched offline for long periods of time, or cannot wait for software to be recompiled once changes are made [3]. These systems include critical high-availability systems such as control systems and many less-critical but still soft real-time, data-oriented systems such as telecommunications systems and network management applications. An infrastructure that supports software hot-swapping must take into account many factors. There are synchronization and timing issues such as when an upgrade can occur and the maximum time window allowed for an upgrade. The size and definition of the incremental swap unit or module must be defined. The series of transactions required to carry out how that unit can be dynamically introduced into a running system must be defined. The state

of the system must be known at all times. Placing the target system in a state where a swap can occur, capturing the state prior to the swap, swapping the module, restoring the system state, and then switching the system over to the new swapped module must all be handled. A failure recovery mechanism must also be in place to rollback an unsuccessful swap without affecting the execution of the running system. System performance must not be compromised and additional side-effects of the swapping process must be minimized. We have developed a prototype infrastructure to support this approach to dynamic software upgrading which takes into account the aforementioned issues. It is described in more detail in another paper [3]. Projects such as SOFA/DCUP [9] have also provided infrastructures to support dynamic component updating in running applications.

2.2 Motivation

While component composition at runtime in high-availability systems poses some interesting challenges, it is focussed on a specific type of system. Many systems cannot be classified as high-availability. For this reason, a more generic composition infrastructure that is not over-complicated with the difficult timing, synchronization, and transactional concerns of a hot-swapping solution would be more appropriate.

The composition architecture described in this paper is dedicated to the creation of composite network services from service components. The research is motivated by that fact that in many areas of network computing the need for more complex services is rapidly increasing. As standardized means for service lookup and deployment become available, the ability to compose composite services out of service components becomes more realistic. The rest of this paper is devoted exclusively to defining and developing a tailored, dynamic service composition solution.

3 Dynamic Service Composition

Dynamic service composition differs from other forms of software composition since it deals exclusively with network services. Network services are individual components, which can be distributed within a network environment, that provide a specific set of well-defined operations. There are two main alternative approaches to dynamic service composition. In the first approach, multiple service components communicate with one another as separate entities to provide an enhanced service that is accessible through a common interface. In the second approach, service components are combined at runtime to provide this enhanced service as a single, self-contained entity. The choice of “communication” or “integration” must be made at the time a composite service is requested by the user. Our architecture will provide some support for automating this choice based on the requirements data collected for a particular composition scenario.

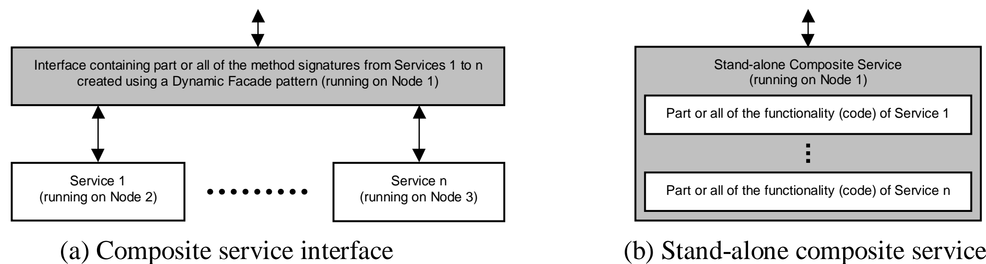
The first step in creating any composite service is to locate the service components that provide the functionality that is to be placed in the new service. To facilitate this process, all service components must be stored in a component directory that can be accessed at runtime. Searches of this directory must be tailored to the compositional attributes of the components. In other words, each component must have a clear description of the operations it can carry out, what methods (if any) can be extracted from it or used in the creation of a composite, and the input and output requirements of the component. Once the appropriate service components are located, we must determine the type of dynamic composition we will perform. There are various tradeoffs associated with the selection of a particular composition method. In many cases, more than one method is possible. However, the selection of the best method should be based on how the composite service will be used and the efficiency requirements of the resulting service. Another objective is to minimize unanticipated service behaviors once the composite is complete.

3.1 Creating a Composite Service Interface

The following approach can be used if a high-level of software performance for the service composite is not required. The idea is to create a new interface that will make a set of collaborating services appear as a single composite service. We have made use of an extended Facade design pattern [4] to help us create this interface. The Facade pattern is intended to provide a unified interface to a set of components and handle the delegation of incoming requests to the appropriate component. However, this is done statically

at design-time. Our Dynamic Facade pattern facilitates the updating of the composite service interface as components are added at runtime. If the services taking part in the composite service are not co-located on the same network node and are instead distributed throughout the network, messages will need to be sent between the components via the interface. We are currently examining the potential for a distributed Dynamic Facade pattern to delegate incoming requests to the appropriate service components even if those components are not located on the same network node. In a distributed composite service, we would expect a slight decrease in response time or operational performance. Figure 1a illustrates the realization of a composite service interface.

The primary advantage of this technique is the speed at which a composite can be created. This is due to the fact that a new component does not need to be constructed. In other words, no code needs to be moved or integrated from any of the components involved in order for the composite service to function. This technique is also referred to as *interface fusion* since the interfaces of each service component involved are merged into a single new interface. However, the interfaces of services 1 to n cannot simply be "glued" together. Modifications will be required to properly direct incoming messages to the appropriate components and in the proper sequence.



(a) Composite service interface (b) Stand-alone composite service

Figure 1: Composite services

3.2 Creating a Stand-alone Composite Service

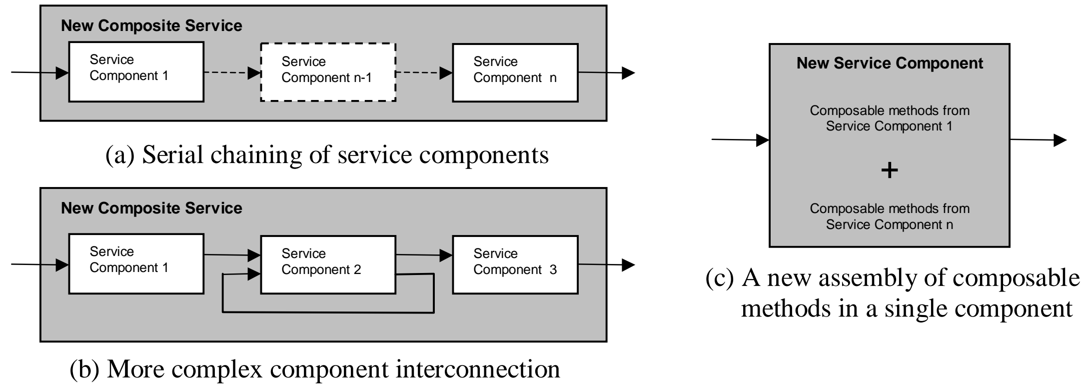
If the performance of the composite service is more critical, creating a stand-alone composite service is a better solution than interface fusion. Performance may be improved since all of the code of the composite service is located on the same node (see Figure 1b). There are two primary means of creating a stand-alone composite service.

One approach leads to the dynamic assembly of service components in a way that is analogous to an assembly of pipes and filters. Jackson and Zave also adopt this technique in their distributed feature composition (DFC) architecture [6]. As in DFC, our architecture has the typical advantages of the pipe-and-filter architectural style. The main advantage is all service components can remain independent. This means they do not need to share state information and they are not aware or dependent on other service components. They behave compositionally and the set of service components making up a composite service can be changed at runtime.

Figure 2a shows a basic configuration for a set of service components to be assembled. The input to the composite service is sent to the first service component, which in turn, sends its output to the input of the next service component in the chain. Obviously, each service component must be capable of handling the input it is given. A different result may be obtained if the components are re-ordered. The order in which components are assembled and the input requirements and output results of each component are specified in the service specification of each component. This service specification is physically stored with the component since the infrastructure will need to read it prior to determining if it will be required in a given composition scenario. Figure 2b shows the potential for more complex interconnections of service components. In this case, the operations performed by service component 2 are required several times in sequence. A loopback data flow can be used to achieve this without the need to chain several replications of the same component in sequence. Support for a loopback feature must be provided by the service component. This capability is also documented in the service specification of the component.

The second approach to creating a new stand-alone service is shown in Figure 2c. Here, the service logic, or code, of each service component is assembled within a new composite service. In general, all of the code from each component cannot be reused since certain methods are specific to an individual service or are not useful in the context of a composite service. For this reason, composable methods are identified

in the service specification of each component. The appropriate sections of the service specifications from each component involved are also assembled to form the service specification of the composite service. The runtime creation of a new and functional service specification ensures that this new composite service has all of the basic attributes of any other service. This upholds the widely accepted principal that the composite should itself be composable [12].



(a) Serial chaining of service components

(b) More complex component interconnection

(c) A new assembly of composable methods in a single component

Figure 2: Fig. 2a and 2b show various potential pipe-and-filter assemblies of service components within a stand-alone composite service. Fig. 2c illustrates an assembly of composable methods from several service components into a single new service containing a single body of code.

The primary advantage of a stand-alone composite service is it can be reused and composed easily with other services. Reuse of a composite service interface is more difficult since the service components providing the functionality are not contained in a single entity. Another advantage is the new composite service will execute at a higher level of performance with regard to internal message transmission since all of the code is executing in the same location.

Constructing a stand-alone composite service at runtime is a very complex undertaking. While many of the processes common to both forms of dynamic service composition are still present (refer back to section 3), other challenges exist. The largest of these is to create a new functional service and successfully deploy it in a relatively short period of time. While the process of combining runtime services could be performed prior to when the service is actually needed, with the composite stored in a library for future use, we are more interested in determining to what extent the runtime construction of a composite service for immediate use is feasible.

Now that we are familiar with the terminology and processes of dynamic service composition, we can determine the requirements of a service composition architecture that will support the creation of such services.

4 Proposed Architecture

We stated earlier that our goal is to design an architecture that makes use of existing technologies. We can justify this choice, over implementing a proprietary solution, since our prototype will not support components that were not originally designed to function as part of a composite service. This does not mean that all possible compositions have to be envisioned before the component can be designed. We still allow the content of the composite service to be determined at runtime. We simply define a set of requirements that each service component must satisfy in order for it to be used in our architecture.

4.1 Basic Requirements

The most critical element of a composable service component is the service specification. The service specification contains the inputs, outputs, dependencies and constraints of the service, in addition to a detailed description of the operations it performs. Another important feature is the composable service component must be easy to locate and retrieve. This is also facilitated by the service specification, which is

examined by the infrastructure during service lookup and retrieval. Finally, the code of each service component can be fully reusable or partially reusable. As we mentioned earlier, all composable methods must be clearly specified in the service specification of the component. It will be impossible for the infrastructure to determine at runtime if code within a service component can be reused unless it has been explicitly labeled as reusable.

An architecture supporting dynamic service composition must have a repository or library of composable service components. This library must allow a service, matching a well-defined set of attributes, to be retrieved in a relatively short period of time. To achieve this, the architecture must have a means of examining the service specification of a service component. This will require that the service specification be written using a well-structured description language to allow for straightforward parsing of the specification file. Finally, the architecture will require a valid component model in order to support component composition.

4.2 Selection of Available Technologies and Required Extensions

Technologies are currently available that can be used to create an architecture to support dynamic service composition within a service domain. However, many of these technologies provide only a partial solution and must be extended and customized to work under the conditions outlined in this paper – conditions which were not necessarily anticipated during the design of these technologies.

We have purposely avoided developing or using a compositional language similar to Lava [7] or CHAIMS [1] since these languages are more appropriate for solving the problem of generic software composition. A compositional language facilitates the following activities: it allows components to be defined within existing non-segmented code, it allows the state and behavior of a component to be inherited by another component, it allows components to be dynamically adapted, or it allows components to be modified at runtime [2]. However, we are not interested in "componentizing" a monolithic software system. This is a completely different body of research with challenges that we feel will hinder our particular interests in dynamic composition of components. Our approach does not facilitate the restructuring of a previously designed system into smaller, more manageable components but rather allows a new system to be designed so it can exploit the advantages of a dynamic component-based architecture.

We have chosen to use Jini connection technology [11], developed by Sun Microsystems, as our service repository and retrieval system. We chose Jini because it is a distributed computing technology that facilitates the lookup and deployment of service components by providing the necessary networking infrastructure and distributed programming facilities. We have created a Jini service called the Composition Manager to oversee all aspects of dynamic service composition.

We assume the reader is familiar with the basic concepts in the JavaBeans component model. However, we will briefly describe the key features of the Extensible Runtime Containment and Services Protocol (ERCSP) for JavaBeans [10] since it provides the facilities we will use for dynamic composition. The ERCSP standard extension provides an API that enables Beans to interconnect at runtime. It enables a Bean to interrogate its environment for certain capabilities and available services. This allows the Bean to dynamically adjust its behavior to the container or context in which it finds itself. The API consists of two parts: a logical containment hierarchy for Beans components and a method of discovering the services that are provided by Beans within such a hierarchy. The containment hierarchy enables grouping of Beans in a logical manner which can easily be navigated. This grouping is established through the use of a BeanContext container. A BeanContext can of course contain other BeanContexts thus allowing for any arbitrary grouping of components. The Services API within the ERCSP gives Beans a standard mechanism to discover which services other Beans may provide and to connect to these Beans to make use of those services. Beans can use introspection to find each other's capabilities.

Figure 3a shows how a JavaBean can be nested within a Jini service to create a service component. In this way we can use Jini for component storage and retrieval while taking advantage of the compositional features of the JavaBeans component model. We have shown the service specification in this diagram to highlight the enhancements we have made to the JavaBean. Figure 3b shows how a BeanContext can be introduced from one service component into another service component at runtime to create a stand-alone composite service. Figure 3c shows that a more reusable stand-alone composite service can be created where all of the code is contained within a single JavaBean.

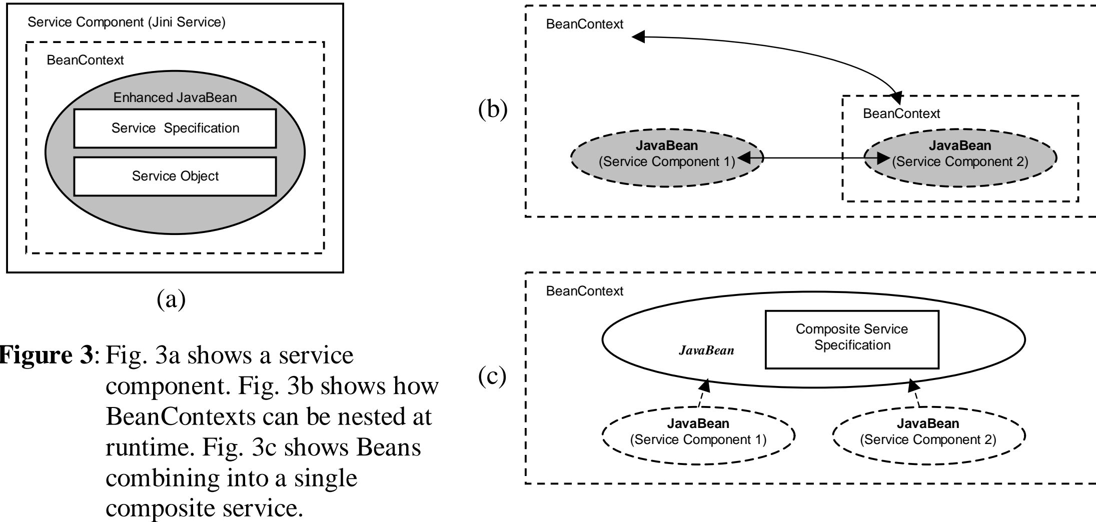


Figure 3: Fig. 3a shows a service component. Fig. 3b shows how BeanContexts can be nested at runtime. Fig. 3c shows Beans combining into a single composite service.

The eXtensible Markup Language (XML) is used to format data into structured information containing both content and semantic meaning [5]. XML provides a convenient and highly effective way to encode a service specification in such a way that the Composition Manager can quickly determine the attributes of that service and the operations it can perform. Figure 4a shows how we have enhanced the Jini Lookup Service to include an XML parser. This will allow us to read the service specification stored in each service component at runtime. A limited example of an XML service specification is shown in Figure 4b.

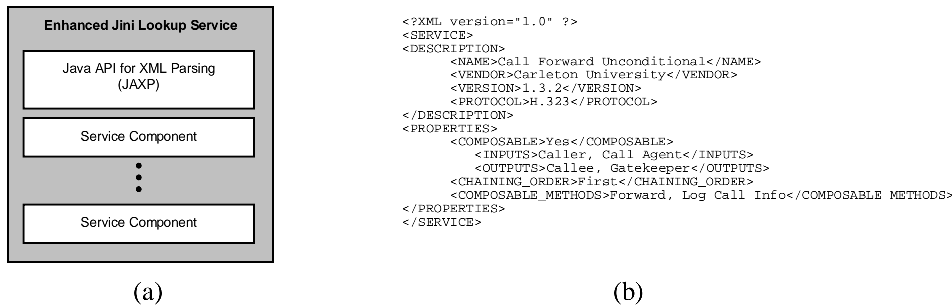


Figure 4: Fig. 4a shows the addition of an XML parsing facility to the standard Jini Lookup Service needed to interpret service specifications. Fig. 4b shows a simple XML service specification.

5 Applications

There are many potential applications for our proposed dynamic service composition architecture. Of particular interest to our research group is the creation of Internet Protocol (IP) telephony services and Intelligent Network (IN) services within the Public Switched Telephone Network (PSTN) [8]. Dynamic service composition has created exciting opportunities for the development of a new service architecture for these telecommunication infrastructures. IN services are designed to add new service logic and data to the existing forms of switched telecommunication networks. The IN platform provides greater flexibility for service creation and allows services to be customized to suit the exact requirements of a particular customer. IN-based services rely on service-independent building blocks (SIBs) which are the smallest units in service creation. SIBs are reusable and can be chained together in various combinations to create services. They are defined to be independent of the specific service they are performing and the technologies used in their realization. SIBs were not originally designed to take advantage of object-oriented concepts and this is one area where the Jini and JavaBean-based service components, described earlier, could be advantageous. Another enhancement our architecture will provide over a SIB-based implementation is runtime component assembly and deployment. Decisions on which service components

will be assembled are made dynamically based on user requirements and are not predetermined at design time as in the SIB approach.

It is a commonly held view within the telecommunication industry that IP-based networks will not replace the PSTN in the short-term [8]. The gradual migration towards IP networks, however, will require hybrid services that can operate in a variety of network environments. In order to accelerate the integration these new networks with the existing infrastructure, services may need to be provided by a variety of parties including the vendors of the network equipment. Vendors are aware of the issues related to protocol convergence and therefore will be involved in the development of the majority of hybrid services. The architecture proposed in this paper allows services developed by vendors, service providers, and individual users to be integrated together assuming they are compliant with a basic set of agreed upon compositional requirements.

5.1 Example

In order to illustrate the types of services that are enabled by our architecture, we will describe a composite IP telephony service that makes use of data, multimedia, and e-commerce service components. A single service provider will deploy this composite service to the display of a telephone handset or wireless phone. This service will allow a customer to use their phone to make a hotel reservation in a city with which they are unfamiliar.

In this scenario, a customer interacts with a reservation service, which is downloaded by the hotel company, over the Internet using the phone. The user proceeds to search for a hotel in the city of interest. Once a hotel is found, the customer is asked if they are interested in seeing a map of the area surrounding the hotel or a list of attractions. This information may or may not be provided by the hotel company itself. If another service provider is able to offer these services, the user may want to use them without having to go through the hassle of signing up for each service individually. Ideally, the user is not even aware that a third party is providing the service.

We assume that before these additional services can be provided, the hotel company will have to make agreements with a set of service component providers to gain access to certain types of information components. These information components could then be used to create a composite service that would be deployed to the customer. Alternatively, the customer could subscribe to a travel service that would provide a list of additional information services that the user could access. In the later case, one possibility would be for the dynamic service composition architecture provided by the travel service to deploy a composite service interface, involving the information components requested by the user, to the phone set. If large amounts of multimedia data are required, as would be the case with an interactive map, a stand-alone composite service may be created and deployed to the phone set instead. If the customer wants to make a reservation with the hotel or pay a deposit on the room, a secure e-commerce transaction component may also be a part of the composite service that is ultimately deployed.

6 Conclusions and Future Work

This paper presents two approaches to dynamic composition of service components and a proposed software architecture to support these techniques. A composite service interface can be created if the composition needs to be carried out in a relatively short period of time. However, there is limited reuse potential for this service since the service components involved in the composite service may be distributed on several network nodes. The advantage of interface fusion is that it is relatively straightforward to assemble the interface and deploy the composite service.

The second method is to create a stand-alone composite service. This is considerably more difficult since code must be physically moved from one component into another. It takes a much longer time to construct a stand-alone composite service since the service specification must be created from the service specifications of the member components and a completely new component must be assembled. The advantage, however, is that this new service can be stored in a service component directory for future use. It is a valid service component just like its member service components and therefore has reuse potential.

Currently, a prototype of our composable service architecture is under development. We are instrumenting our system with performance metrics and plan to carry out a scalability analysis in an effort to quantify the limitations of our proposed solution. We realize that scalability is already an issue with Jini

since the technology is targeted to a network (or workgroup) of one thousand nodes in size. Therefore, the applicability of Jini in larger networks is doubtful. However, this problem can be handled to some extent with a hierarchical infrastructure organization. Once our prototype implementation is complete, we will focus on developing applications of dynamic service composition such as IP telephony and multimedia service composition and the runtime assembly of new network management services.

Acknowledgements

The authors would like to thank Babak Esfandiari and several other anonymous reviewers for their helpful comments. The research in this paper is supported by Communication Information Technology Ontario (CITO).

References

[1] D. Beringer, C. Tornabene, P. Jain, G. Wiederhold, "A Language and for Composing Autonomous, Heterogeneous and Distributed Megamodules," International Workshop on Large-Scale Software Composition, in conjunction with the 9th International Workshop on Database and Expert Systems Applications (DEXA '98), Vienna, Austria, August 1998, pp. 826-833.

[2] J. Bosch, "Superimposition: A Component Adaptation Technique", Information and Software Technology, Vol. 41, Issue 5, March 25, 1999.

[3] N. Feng, "Software Hot-swapping Technology Design", Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University, June 1999.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[5] S. Holzner, XML Complete, McGraw-Hill, 1998.

[6] M. Jackson, P. Zave, "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services", IEEE Transactions on Software Engineering, Vol. 24, No.10, October 1998.

[7] G. Kniesel, "Type-Safe Delegation for Runtime Component Adaptation", Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99), R. Guerraoui (Ed.), Lisbon, Portugal, June 14-18, 1999.

[8] B. Pagurek, J. Tang, T. White, R. Glitho, "Management of Advanced Services in H.323 Internet Protocol Telephony", Proceedings of 19th Annual Conference on Computer Communications (Infocom 2000), Tel Aviv, Israel, March 26-30, 2000.

[9] F. Plasil, D. Balek, R. Janecek, "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98), Annapolis, MD, May 4-6, 1998.

[10] Sun Microsystems, "Extensible Runtime Containment and Services Protocol for JavaBeans Version 1.0", L. Cable (Ed.), December 3, 1998.

[11] Sun Microsystems, "Jini™ Technology Starter Kit Documentation Package", Version 1.0.1, November, 1999.

[12] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.