Context Federations:
A Virtual Context for Software Components

Micha Moffie Wenxu Tong David Kaeli David H. Lorenz

Northeastern University
Boston, Massachusetts 02115, USA
{mmoffie@ece, wxtong@ccs, kaeli@ece, lorenz@ccs}.neu.edu

Abstract. In this paper we introduce the concept of *context federation* that extends the traditional notion of a context in the domain of software component design to include a composition of several physical contexts. We extend the component model to support context federation, and describe a new context composition model which we call *federal composition.* We illustrate the model by designing an infrastructure for the JavaBeans technology that implements our context federation model and supports federal composition. We discuss applications of a context federation to illustrate its usefulness.

1 Introduction

Scalability of Enterprise JavaBeans (EJB [10, 9]) servers is becoming increasingly important. While more and more transactions are processed concurrently, the system as a whole must provide for performance and dependability. A common solution is to create clones of the EJB container and execute them concurrently. Federating and coordinating these containers (contexts with services and components) then becomes a new problem [2].

Consider the case where an EJB service is modified. The service has to be updated in all contexts at once. Similarly, when a new service or component is introduced into one of the EJB containers, that service or component needs to be available to all other EJB servers. IBM WebSphere, for example, provides a single administrator facility to manage all WebSphere servers.

In this paper, we propose a solution to alleviate this management difficulty by introducing the concept of a *virtual context.* A virtual context is a federation of contexts that may span several physical locations, yet provide a single logical context for all components and services.

We develop a new *federal composition* model that extends the manner in which components can interact and services can be acquired. Components residing in different physical contexts may be composed as if they logically reside in a single context. Components of a logical context can easily acquire services from a remote physical context as if it were local.

We present a composition framework that supports federal composition. The framework supports the the merge of several physical contexts into a single logical context. We present an infrastructure to support our context federation model in the JavaBeans component model.

In the rest of the paper we will make use of the JavaBeans terminology. We use the term *context* to denote a JavaBeans context—a *container* that may have component instantiated inside of it and may also be itself a component of an encapsulating context. A *parent* of a context is defined as the container of the context. A *service* is a reference to an object that may be obtained by the component when interrogating its context. A software component is "a unit of composition with contractually specified interfaces and explicit context dependencies only" [7]. A component framework is "a collection of rules and interfaces (contracts) that govern the interaction of components plugged into the framework." [7].

1.1 Component composition frameworks

Current component frameworks emphasize two main composition models. The main difference between models is the way components and contexts interact. Some composition models emphasize the interaction between components and their contexts, while others accentuate the interaction between the components themselves.

Contextual composition Contextual composition is the composition of components with respect to a context that has a set of properties on which all the components agree [7]. An example of such a context would be a context that has synchronization properties. Enterprise JavaBeans is a good example for this kind of composition, since EJB uses the automatic composition of component instances, and appropriate services/resources.

Connection-oriented composition The JavaBeans approach to composition is different than the one used by EJB. The JavaBeans approach uses a connection-oriented approach (wiring) [7]. Components are connected using events; components interact by sending and receiving events.

The contextual composition model places an emphasis on the interaction between a component and its context. The connection oriented composition model, on the other hand, emphasizes the interaction between the components themselves.

1.2 The JavaBeans Component Model

In the JavaBeans component model, the user defines event sources and event listeners. At design time, beans are wired together. A wire can extend from sources to listeners and represent a communication channel. At run time, the wired component interact via events.

The JavaBeans model provides standard mechanisms for a component to obtain "context-specific facilities or services from its environment" [1]. For example, a bean can interrogate its context to see if a printing service is available.

If such a service is available, the a reference to a printing service instance is returned.

A JavaBeans context (BeanContext) is comparable to the EJB container—it holds the components and is responsible for providing services needed by the components. The JavaBeans model allows us to create a hierarchical context structure. Thus, there is no need to place all of the JavaBeans in a one flat container [7].

We use the BeanContext model as a reference model, and build our models on top of it. The BeanContext model has the following characteristics:

– All components are local.
– Each component has at most one parent.
– Logical equivalence is physical equivalence.

Physical equivalence is defined as an exact copy of an object. We can define it as the exact same address in the memory (on the same machine), or as the exact same representation in memory. *Logical equivalence* is defined as components that have matching signatures and conforming functionality.[1]

The JavaBeans context model imposes a constraint that every component in the system must be local to the system and must be both physically and logically distinct from other components in the same system [1]. Context-membership relationships are fundamentally a tree structure (Figure 1). All the JavaBeans are organized in a tree structure.

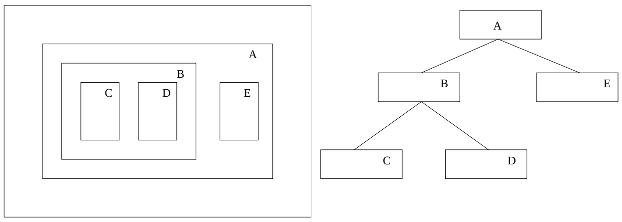

Fig. 1. Containment relationship in the JavaBeans context model can be represented as a tree structure. Every component is represented as a node. The components C and D are contained as children of the context component B, which is contained in the context object A.

A BeanContext may or may not have contained components. A contained component is either a simple bean that does not require support or knowledge of its environment or a nested BeanContext or BeanContextChild object. A JavaBeans component will require support or knowledge of its environment if it needs to request services. In order to obtain services, a nested component will need to query its parent. The parent can either return a reference to an object providing the service (if it has one) or query its own parent for the service. Note

1 *Physical equivalence* implies *logical equivalence.*

that each BeanContext has only one parent containing the BeanContext. An exception is the top most BeanContext, which does not have a parent.

Outline. This paper is organized as follows. In the next section we describe the federal composition model in more detail and the context federation model. This is followed with a section describing an infrastructure design for implementing the context federation. Next, we detail how two example scenarios can be implemented in our context federation framework. In the last section we conclude the paper.

2 The Concept of a Context Federation

In this section we explain the federal context composition model, and distinguish it from currently known composition models. We then formalize the notion of context federation. We show how a context federation can be constructed and discuss some limitations.

2.1 Federal Composition

Federal composition is a distinct composition model for composing context components. The contextual and connection-oriented composition model described above emphasize aspects of interaction between components themselves and between components and services that are related to the same context (or container). The federal composition model goes one step further and introduces a composition between different contexts.

By composing different contexts, we are able to introduce additional means of interaction between different elements associated with the different contexts. Federal composition allows components and services from different contexts to interact as if they belong to the same context. Such an interaction may result in exchange of services (between the contexts), such that the components residing in the contexts are introduced to additional services. An example of this would be the composition of two contexts, one providing a printing service. Although the service is supplied by only one of the contexts, the composition of those two contexts allows components residing in both contexts to ask for the printing service—as if it were provided by the native context.

Another benefit of federal composition is the interaction between components in different contexts. A federal composition of contexts allows components of originally different contexts to interact as if they reside in one context. This does not necessarily encompass all the components in the composed contexts. The model permits to choose which components participate in the composition and with do not. Only the selected components will interact.

The federal composition model does not imply any specific semantics on the context composition operation. It allows us to compose contexts in different ways, resulting in a variety of outcomes. Any framework that supports federal composition may define its own composition semantics. Such a framework may also impose different restrictions on the contexts being composed to avoid undesired compositions.

It is important to note that federal composition is not just a simple union of the contexts, nor is it equivalent to creating a new parent that includes all the contexts. Both options might even not be physically possible—if the contexts are running on different machines. Federal composition is best described as a sophisticated merge operation on contexts.

2.2 Context Federation

The context federation model supports federal context composition. Our model limits the composition only to those contexts, components and services that we regard as "legitimate." We restrict the contexts being composed to be logically equivalent.[2] The reason for this restriction stems out of the fact that we aim to create one virtual context. This virtual context represents the composition of several contexts and must be well defined.

Next, we define a specific operation for context composition. The composition operation can be divided into the following two steps:[3]

1. Defining a virtual context (given a set of contexts).

$$Virtual(\{C_1, C_2, \ldots, C_n\}) \Rightarrow V^*$$

2. Establishing the contexts federation. The federation is a realization of the virtual context.

$$Federation(V^*) \Rightarrow \{C'_1, C'_2, \ldots, C'_n\}$$

where $\{C_1, C_2, \ldots\}$ is a set of contexts which are being composed. V^* is the virtual context; and $\{C'_1, C'_2, \ldots, C'_n\}$ are the resulting set of modified contexts—collectively making up the context federation. The second step actually modifies, updates or replaces the original contexts with a representation of the virtual context. The resulting contexts are then said to be *equivalent*.

The virtual context represents several equivalent contexts, and is used to define the semantics of the composition operation. We call it *virtual* because the corresponding contexts are logically equivalent, as if only one virtual context exists. In other words, the context federation is a set of logically equivalent contexts.

2.3 Defining a Virtual Context

When defining a virtual context V^*, we need to consider the following elements that take a part in the composition: components, services and the context instance itself (the context as a component).

[2] Note that this restriction must hold for the contexts being composed but it is not mandatory for the components and services.

[3] Both steps are presented here for clarity; an implementation may not make such a clear distinction.

Both the components and services in each context can be classified as one of the following:

Shared. This kind of component or service is virtual. It is available in all contexts.

Local. This kind of component or service is not virtual. It is private to this context and not part of the virtual context.

Shadowed. This kind of component or service is virtual. It is a part of the virtual context. Some contexts may have a local implementation (for the component or service) that is logically equivalent and will be used locally.

The components of the virtual context include the *union* of all the shared components and a *merge* of all shadowed component (across all participating contexts). The local components are discarded. The union is a simple addition of the shared components in the contexts, while the merge collapses two or more logically equivalent components (from different contexts) into one logical component in the virtual context. For example, we can create a virtual context from the following two contexts: the first includes shared components A and B, local component P and one shadowed component R1. The second includes one shared component C, one local component Q and one shadowed component R2 (that is logically equivalent to R1). The virtual context that is created will include components A, B, C, and R which represents either R1 or R2.

The Services are composed in a similar way to the components. The virtual context will include a *union* of the shared services and a *merge* of shadowed (logically equivalent) services. The local services are discarded.

The contexts themselves must have matching signatures and conforming functionality, i.e. be logically equivalent as objects (discarding their subcomponents or services).

2.4 Constructing the context federation

The federal context is, as mentioned above, the realization of a virtual context. A realization can be implemented by updating or modifying the context (adding components and/or services), or by replacing it with a new context, which is part of the federation.

When creating the federation $\{C'_1, C'_2, \ldots, C'_n\}$, we first need to consider each of the resulting contexts: C'_x. The resulting contexts will become a reflection of the virtual context. They will include components and services, some original (local, shadowed) and some new (shared, originated from other contexts).

The components in the resulting context will include: the original local components unchanged, the original shadowed components unchanged and all the shared components that are in the virtual context. Continuing with the example presented when creating the virtual context, the federation of the two contexts will include the following modified contexts: The first context will include: shared components A, B and C, local component P and shadowed component R1. The second context will include: shared components A, B and C, local component Q and shadowed component R2.

The resulting contexts are now logically equivalent. During run time the federation itself, and the virtual context may be updated: components may be created, added or removed and services might be modified. It is the responsibility of each of the contexts (or their components) participating in the federation to update the relevant shared components and/or services. This updating is needed to constantly reflect the same logical state across all contexts participating in the federation.

Limitations. This model restricts the composition of contexts only to contexts that are logically equivalent. We argue that this limitation is lenient since in most cases creating a context federation implicitly means that the contexts themselves and many of the components and services are already logically equivalent. There is no sense in creating a federation where there is no common data or behavior.

3 Context Federation Frameworks

In this section we describe several frameworks for establishing a context federation. Each framework implements a different (potentially limited) model of a context federation.

We start with the simpler and move to the more complex and elaborated ones. We illustrate by example the usefulness of each of those frameworks. We then build upon it for the next framework.

The goal of context federation is to create a virtual context that can be realized on several hosts. The difference between the frameworks lies in the way they define a virtual context, i.e., what type of elements are being *virtualized* and how.

3.1 Virtual Components

We begin with the simplest model. In this model, the federal composition allows us only to share components between contexts, i.e., only components are being virtualized. The contexts being composed are not necessarily aware of that their containing components are virtual.

There are two implementation strategies, namely, a *proxy-based* virtual component model and a *delegate-based* virtual component model.

In the proxy-based virtual component model we have a local proxy/interface to an actual component on the remote side. Every service request to that component is sent through the proxy. The same mechanism is used for obtaining the result back (Figure 2).

This is the simplest and the most straightforward virtual component design. Applying this design to the JavaBeans model, the only advantage gained in this extension is that now we can include a remote component in a local context and use its services in the local environment. This allows local BeanContext components to interact with the virtual context. This model is useful in some simple but useful applications like printing a local file on a remote printer.

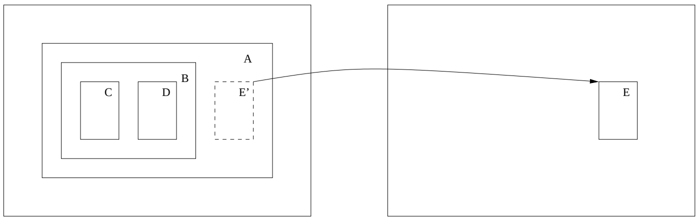

Fig. 2. proxy-based virtual component.

In the proxy-based model, the environment of the remote component (parent) may not be changed to the local containing context. The result is that the remote component can interact only with the remote environment. Thus, a local component interacting with the remote proxy must take into account that the local environment cannot be used. For example, assume a virtual (remote) component that implements a postscript driver and can print on any printer that is accessible via the remote context (its parent). In this case, printing a local file on a remote printer causes no problem, but using the remote component to print a postscript file on a local printer cannot be done. So the interaction is only one-way. Only virtual components that are oblivious of the environments of the local context containing them fit into this model.

To enable interaction with local environments, we propose an alternative approach: delegate-base virtual component model. In the delegate-based model the component is being copied to the local context. Since the exact copy includes a reference to the containing context (the parent field) we must consider two cases: in the first case, if the parent of the original virtual component is NULL (no parent), meaning the virtual component being copied does not require interaction with the environment, it is readily integrated in its local context. In the second case if the parent field is not null, the parent value must be set to the local context object that now contains the component.

The virtual component models introduced above enables a local context to contain a remote component almost as if it were local. Our virtual component models allow us to create a logical context only on one site, using virtual components from remote sites. This introduces an asymmetry: only one site has the containing context, while all other sites only instantiate components and have no access to the containing context.

3.2 Simple Virtual Context — federating the components

In order to take full advantage of the BeanContext model in a distributed environment, a full virtual context model is needed. In this section we address the issues of creating symmetrical models of real virtual contexts, which allows not only the component but also the containing context, to be virtualized.

As stated above, when federating two or more virtual contexts we need to address a few issues on how to treat the parents (services), subcomponents and the virtual context themselves. Those issues will impact the behavior of the context in each of the realizations (local views) of the virtual context.

The most straightforward way to create a virtual context from a set of different, but logically equivalent contexts is to unify all their subcomponents (and only the subcomponents). Thus the virtual context will include all the subcomponents but no services. In this case, each local context keeps its original parent and original environment, while having the same set of subcomponents.

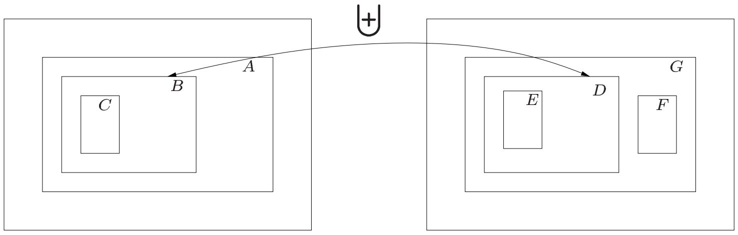

Fig. 3. Simple Virtual Context model. Before federation.

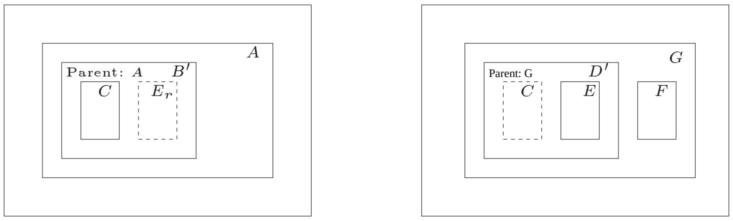

Fig. 4. Simple Virtual Context model. To federate two contexts of Figure 3 (B and D), remote components (E' and C') are created into the local context, through either reference or copy. The two resulted contexts B and D still keep their original parents. (A for B and G for D).

A good real-world analogy would be the merger of two Departments of Computer Sciences that originally resided in the Engineering College and in the College of Arts and Sciences. The merger should enable everyone in the two departments to share resources. Both departments have their own resources as well as the resources that they inherit from the associated college. Note that we do not want to merge the two colleges nor do we want to merge their resources as a result.

We would like each student from the Engineering Department (after the merge) to be able to access all the resources used to belong to the Arts and Sciences Departments as well as those from the Engineering College. We would like the students from the College of Arts and Science to be able to access all the resource that belongs to the Engineering College plus what originally accessible from the Arts and Science College. Also, the properties belonging to the original departments still remain unchanged for the departments.

The simple virtual context models introduced above allows several contexts to share a set of components while keeping the realization of this set in their respective local environments. We allow all the components to seemingly interact with each other as if they reside in one component. At the same time those component use the services provided by each local context.

3.3 A Full Virtual Context

In this section we introduce the most complex and flexible context federation framework. The framework allows the virtual context not only to have different subcomponents as well as services, but also allows each context to choose which subcomponents and services should be local, shared or shadowed.

This model allows each of the logically equivalent contexts to use a locally defined component to shadow (similar to the override in traditional OO) a globally defined corresponding component, if they have conforming signatures.

In this model, a locally defined copy of the component may have the same interface/signature as the remote component. But the implementation of each service can be either same or different to the remote one. Also, properly defined, even the membership can be the same or different. Local membership or service can shadow the signature-matching remote ones. This way, each logical context provides some kind of interface or super class with some default implementation, while each physical copy is a concrete implementation.

This design gives the virtual context model great flexibility. When a new component is added to a context, the user can specify one of the following kinds:

- a global/shadowable component, which allows other corresponding logically equivalent component in another local realization of the virtual context to access and shadow it; or
- a locally used, independent, and non-shadowable component.

When unifying two contexts, the contained member components are federated in this way, if a shadowable component in one of the virtual contexts does not have a conforming counterpart defined in another federated context, a copy or a proxy to the component will be created in that context. If both federating contexts have a global/shadowable component signature-matching to each other, both keep their own local copy after the federation. If the component is non-shadowable, then no matter whether there is a conforming signature component in the other federating context, no remote copy of this component will be created in the other context.

Using this model, we can provide a mechanism that within a virtual context, any services provided by a containing remote component can be accessed or shadowed with a possible local implementation with possible local features. Because these features can either be stored in a property field of this context, or as a shadowing member component that shadows a component provided by the other context. At the same time, a component without local specific features can be declared as a shadowable component where global features and services are provided to another federating context if there is no such feature or service defined there. On the other hand, if the whole purpose of a component is for local use and it is not designed to be used anywhere else, it should be declared as a non-shadowable component. But this flexibility comes with a price that the behavior consistency may be lost in this model.

The global/shadowable and local/unshadowable is similar like the remote services and local services in EJB. Both global/shadowable components in our model and remote services are designed to be usable in another context, while the local/unshadowable components and local services are only for use in this context. However, in the EJB model, every service has its own name; and an exact name and a naming service are needed to discover and invoke the service. Our model provides an overriding mechanism of service discovery that is not part of the EJB model.

The global/shadowable and local/unshadowable feature are similar to the virtual and non-virtual methods in the C++ programming language. But it is more flexible than typical inheritance in object-oriented languages. In a typical inheritance case, a subclass inherits a superclass by inheriting all the superclass's features (attributes and methods). In an object-oriented design, if there is a group of related subclasses, a typical design would abstract their common features and put that into a superclass, letting subclasses inherit from the superclass. All common features have to be implemented in the superclass, and special features reside in each subclass.

The object-oriented model is not symmetric in a sense that the superclass is somewhat unique to the subclasses. But in the total federation context model, the context can be totally equal in the sense that each remote context component can have its own global/shadowable and local/nonshadowable properties and services. If it is declared as global/shadowable, then it can be accessed and shared by all its peer contexts of the same kind in another environment, as long as the later does not have a corresponding local property or service shadowing it.

A potential inconsistency may arise in this model if we try to federate more than two contexts. Let us consider if we want to federate three contexts. If a shadowable components are defined in two of the three contexts, but not the third one, after the federation, which of the two shadowable component should the third context have? We do not have a perfect solution for this problem. But we do propose two possible approaches to resolve this inconsistency. One solution is to treat federate as a binary operation and which is not associative. In that case, the order of the federation would matter. Which of the shadowable components the third context gets depends on the order of the federation. This is a straightforward solution.

Another approach is whenever we declare a component/property/service to be global/shadowable, we specify whether it should be set as default in future unification. The rational is to only declare a component with only global information and generally useful even in other environments to be default shadowable. When federation is performed, it will check from all the conforming signature properties/components/services of the contexts to be federated, there should be at most one default global/shadowable defined. If there are more than one declared default properties/components/services, the user can either change one of the components signatures, or randomly pick one as default, or simply not allow such federation. Either one of the approaches and a lot others can solve the consistency problems.

This model is useful for our chess-playing example. We can create the board configuration component as a global component in one player's context and let the context of another player either have a reference to this component or have a deep- matching copy of this component, while we also have another Display component to display the board according to the board configuration state. This Display component is just a signature-matching pair that provides the same kind of service but through different behavior in two different environments.

4 Implementation

All of the models in Section 3 are implementable using current technologies. In this section, we describe the implementation of the delegate-based virtual component model and the simple virtual context model. To implement this we have extended the Java BeanContext using CORBA (Common Object Request Broker Architecture) as the mechanism to facilitate inter-context communication. We have implemented a simple application in which a BeanContext can contain ColorBean components. Both BeanContext and the ColorBean has a color property, which can be set to different colors, choosing colors from a color palette provided by the BeanContext.

To make the model more interesting, we added a policy that restricts a ColorBean from being contained in a BeanContext if the ColorBean and the ColorBeanContext have the same color. When changing the color of the component or the container causes a conflict between the container's policy or some of its contained components, the color change takes place, but the conflicting ColorBean is expunged from the container (and placed in the parent of the context). Also, adding a ColorBean into a container with the same color is disallowed. We shall now see how this application is implemented using two models: the delete-based and the simple virtual context models.

4.1 Delegate-based virtual component model

In the implementation of the delegate-based virtual component model, we built our ColorBean application on the system for membership synchronization, property uniform and service sharing. In our test, BeanContext1 has two ColorBean components, each has a color property. We constructed a new BeanContext2 and include the ColorBean components in BeanContext1 as BeanContext2's remote

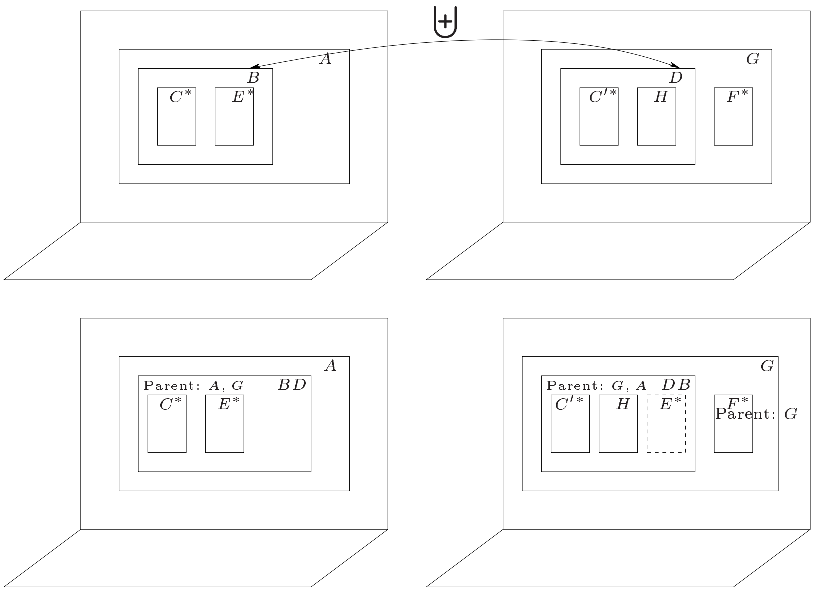

Fig. 5. Total-federation context model. In the above figure, "*" indicates a shadowable component. The character of the component indicates the signature of the component. Context DB and BD after the federation are two physically distinct but logically equivalent components. C^* and E^* are shadowable components in context B. C'^* is shadowable component in context D and it is signature-matching with C^*. H is a local/non-shadowable component in D. F^* is a shadowable component in G. After the federation of B and D, the resulting context BD and DB each keeps their own shadowable component C^* and C'^* respectively. DB contains a remote component of E^* since it is a shadowable component in B and there is no signature-matching component in D. H is not a remote component for BD because it is local/non-shadowable. F^* is not a remote component for BD because we are unifying B and D where F^* is not a contained component is either of them.

components. After the setup, we can change the color property of any ColorBean component from a BeanContext and the change in one BeanContext will be reflected in the other BeanContext as well.

The mechanism to keep the consistence and synchronization of the logical context is implemented as follows. Every component has a Hash-table keeping the copied information about itself. Each component has a flag to indicate whether it can be modified. If the state of the contained component needs to be modified, it will check and lock its status, record its current status, do the modification, and inform the modification to all of it copies. If every of its copies are updated successfully, the change is successfully committed and the lock was reset, otherwise, the modification has to roll back and restart the transaction in next try.

4.2 Simple virtual context model

In our simple virtual context model, BeanContext1 has ColorBean1 and BeanContext2 has ColorBean2, after the unification of BeanContext1 and BeanContext2, both have two ColorBeans. The color property of both ColorBeans can be changed from any of the BeanContext and changing the color property of the unified BeanContext will affect both physical BeanContext components as expected. If we add one more ColorBean component to one of BeanContexts, it will automatically appear in another context. If BeanContext1 and BeanContext2 has two different restrictions for adding new components, the new restrictions of the unified context is the unification of both sets. If other rules for combining restrictions are preferred, they can be easily set. Thus, we achieved the unification of membership, property and service of the two BeanContexts after unifying them.

The implementation for this example is designed as follows. Each of context has a list to keep track of its components. A flag is associated with each component to indicate whether it belongs to local or to the remote site before the federation. After the federation, both contexts will have the components from the remote site added into its own set. [4]

4.3 Lesson

Our implemented models and ColorBean test cases indicate that these virtual context models are implementable. We can associate/de-associate an already existing remote component into a local context. We can change properties of a remote component. To ensure consistency, we think that a component should be created and destroyed locally. However, if for some reason, the remote creation/destroy of a component is a desired feature, it can be easily incorporated into the model with appropriate mechanisms. The remote components and contexts behave exactly as expected and the location of component and context is transparent to users. We can also use federation to merge two contexts into one virtual context.

The ColorBean example shows that after the federate operation, two physically distinct contexts became logically equivalent. There are still some features left out of the models, e.g., conflict resolution in the component federation. One way is to disallow such federation, another would be to expel the conflicting component. We think it is appropriate to leave these issues as implementation policies.

5 Conclusions and Future Work

The virtual context abstraction can be very useful when implementing a distributed application. In many cases different contexts on different machines share

[4] Consistence and synchronization are taken care of through the same mechanism as in the Delegate-based virtual component model.

some state and implement similar elements. The federal composition can alleviate much of the overhead and redundancy associated with implementing such applications. Components and services can be easily shared, duplicated and synchronized. Services that are not locally present can still be used (remotely). Design assumptions and interfaces are more easily kept in sync, and component reuse is improved. Federal composition can be very useful in current composition frameworks. Since it does not conflict with any of the current composition methods, it can be added on top of any such composition framework.

In this paper we introduce the concept of federal composition. We distinguish this type of composition from known composition types. We argue that our composition method introduces new ways of composition, that benefits from additional flexibility and reuse. We describe a context federation framework that supports federal composition. We show how different scenarios can be easily mapped to this framework. We discuss different implementation strategies for supporting the framework and Federal composition, and present a simple framework implementation for federal composition.

Future work include further experiments with our federal context federation. We intend to complete the implementation of all the federal context models, and use the additional experience to refine our model.

References

1. JavaBeans Glasgow specification. JavaSoft. `http://java.sun.com/beans/-glasgow`.
2. J. Sousa and David. Formal modeling of the enterprise JavaBeans component integration framework. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1281–1300. Springer-Verlag, 1999.
3. C. Szyperski. Beyond objects: Components and architecture. *Software Development*, Oct. 2000.
4. C. Szyperski. Beyond objects: Components and contracts. *Software Development*, May 2000.
5. C. Szyperski. Beyond objects: Point, counterpoint. *Software Development*, Feb. 2000.
6. C. Szyperski. Components and continuity. *Software Development*, Apr. 2001.
7. C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2^{nd} edition, 2002. With Dominik Gruntz and Stephan Murer.
8. C. Szyperski. Universe of composition. *Software Development*, Aug. 2002.
9. A. Thomas. Enterprise JavaBeans server component model for Java. Prepared for Sun Microsystems by Patricia Seybold Group, 1997.
10. T. Valesky. *Enterprise JavaBeans*. Addison-Wesley, 1999.