Integrating Container Services with Pluggable System Extensions

Andreas Leicher[1], Alexander Bilke[1], Felix Bübl[3], E. Ulrich Kriegel[2]

[1] Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
{aleicher|bilke}@cs.tu-berlin.de
[2] Fraunhofer Institute for
Software and Systems Engineering (ISST), Germany
Ulrich.Kriegel@isst.fhg.de
[3] Imphar AG, Berlin, Germany
felix.buebl@imphar.com

Abstract. Common middleware platforms support software components by providing a number of standard container services. Adding new container services can adapt a system to changed requirements without modifying its constituent components. Custom-made container services can intercept communication calls between system's components and enforce new or changed requirements. Hence, legacy or off-the-shelf components can be adapted to comply with requirements that are not taken into consideration by the initial system design.
In a global market, companies must be able to quickly adapt their software systems to requirement changes. However, not all middleware platforms allow to integrate additional container services, yet. This paper describes a framework that enables developers to *dynamically* integrate additional container services into existing systems and to configure them at runtime. Moreover, the framework allows for conditional execution of these services – they can adapt to the system's state or its current context.

1 Introduction

Software systems are inherently complex and, in many cases, long-lived. To increase understandability and maintainability, different aspects of a system have to be separated during all levels of system development. The corresponding *'separation of concerns'* problem has been addressed by the research community in recent years, resulting in the development of technologies that facilitate separation of concerns during implementation, e.g. aspect-oriented programming [14] or composition filters [3]. Unfortunately, these technologies require access to the source code or are built on special infrastructures.
In object and component-based middleware, separation of concerns can be achieved by transparently adding services into the middleware without modifying the system's or application's components. Service integration can be realized by two

similar approaches: proxies and interceptors. Each approach allows for executing services if a kind of trigger (e.g. a communication call event) fires. However, interceptors are in some respect more general than proxies as they allow adding services for arbitrary triggers, whereas proxies are focused on distributed communication.
Interceptor-based techniques define the technical ability to add services transparently into middleware. However, complex problems arise when multiple services should be integrated at the same time: Which service should gain precedence and therefore be executed first? Are the preconditions of a service execution altered if other services are executed first? Are service execution chains possible (services triggering each other)?
The purpose of this paper is to enrich interceptor and proxy based approaches with additional aspects dealing with complex service order and conditional service execution:

Complex execution order. Up to now, distinct services can only be executed using simple predefined strategies. Mainly a linear or sequential service ordering and execution is applied. However, so called feature interaction [5] between services require non-linear execution, because services can have side effects according to the component's states or to their context. A non-linear execution order can arrange several services within one communication call. Consequently, a flexible hierarchy of services is needed.

Conditional execution. Sometimes business logic requires conditional execution of services. With regard to service reuse, it is an advantage to separate services and their conditional execution logic: First, services become independent of changes. Second, system dependent properties can be computed by the conditional logic part. Thus, services become independent of the middleware. Third, in conjunction with service ordering, it should be possible to optimize service execution.

Dynamic Configuration. If proxies are used, the additional functionality is often hard-coded into the intercepting code at development time. Thus, it is not possible to reconfigure and exchange these functionalities without stopping and restarting the system. It should be possible to modify a running system with a dynamic configuration mechanism, to adapt it to new or modified requirements like dynamically changing debugging code.

The structure of this paper is outlined in the following. Section 2 describes the terms *proxy* and *interceptor* as a means to intercept method calls and discusses implementation approaches. *Plug-ins* and the *proxy manager*, which are needed for dynamic integration of new functionality, are introduced in section 3. Section 4 deals with the feature interaction problem that arises when concerns have to be combined. The use of conditional expressions is described in section 5. The resulting consequences for practical use are discussed in section 6 on the example of the EJBComplex framework, developed to facilitate dynamic integration and combination of new container services into the EJB platform. The configuration of the framework is explained in section 7, followed by related work in section 8 and the conclusion in section 9.

2 Technical Background and Underlying Concepts

There are two ways to add services to applications running on middleware systems. First, a system level mechanism can be used, which requires the modification of the middleware itself. Second, an application level mechanism can be used, which can be implemented without affecting the middleware implementation

The interceptor pattern describes a system-level integration. An interceptor is defined as a concept to *'transparently integrate services into a framework, e.g. middleware'*[10]. The application of this concept on system level requires the modification of internal functionalities of middleware which means that the middleware's source code has to be modified. At least, the interception call has to be integrated into the middleware code. The advantage of this solution is a completely transparent integration of services that does not suffer from problems related to solutions on application level. However, the necessity to manipulate the source code and the resulting dependency on a certain product is a serious disadvantage.

There exist a number of terms describing the interceptor concept. In some middleware products interceptors are known as callback functions. Callback functions modify standard behavior and simplify integration approaches: e.g. , the JBoss Server [20] supports a callback interface to intercept method calls to Enterprise JavaBeans. The BEA WebLogic Server [2] provides a callback interface in order to specify customized load balancing. Microsoft offers integration techniques, so-called hooks, which easily and transparently allow addition of interception code into existing software [13,17]. CORBA middleware explicitly defines interceptors as a standard extension mechanism.

However, the EJB middleware specification has no such mechanism. Thus, it is necessary to use an application-level concept, such as a proxy. A proxy is characterized by both an identical interface and a definite reference to the encapsulated object. Gamma et al. [11] defines a proxy as follows: *'A proxy provides a surrogate or placeholder for another object to control access to it.'* Since a proxy is called instead of a remote server object, it can be used to prepare a dedicated runtime environment for the remote server object and call it afterwards. On principle, that is how modern application servers work.

At application level, there are some approaches to apply the proxy pattern, represented in figure 1. The standard approach is shown as variant one. It is based on a proxy component, which is transparently placed between the service provider and the client component. The advantage of this approach is that application level components do not have to be modified. However, compared to the same principle on system level, control is limited. For instance, the self-problem (disclosure of the component's identity) cannot be fully prevented. The `Server` component could return a self-reference to the `Client` and, therefore, enable direct communication bypassing the proxy class. Another disadvantage arises because the specification of the middleware allows only restricted access to system properties. For instance, in an EJB environment the caller's identity can not be determined. Variants two and three can be used to merge the proxy function-

ality and the component functionality (`Server`) into a single new component, which prevents the self-problem. The second variant shows the inheritance of the application level component. The original component is specialized and replaced by a new component. The third variant shows a aggregation approach. The original component is encapsulated into a new component that also includes the interception functionality. These variants require the modification of the system because components have to be replaced.
In general, mechanisms to allow service integration are well understood. However, the mechanisms deal not well with problems arising if multiple services are integrated, such as service ordering, conditional execution, and dynamic service modification. This paper suggests structures and modeling solutions dealing with these problems. These solutions are applicable with both system-level and application level integration.

2.1 Extension Services

It is of great importance to carefully consider what kind of functionality will be integrated into a system's environment. Since components are the loci of computation, it should be forbidden to alter a system's application logic outside of them. Present-day middleware platforms provide no direct access or limited access to their communication mechanism's implementations. However, there are non-functional services, which do not influence a system's computation, for example the installation of debug functionality or constraint monitoring at runtime (see section 7).
In the following, we use the term **extension service** to describe services that do not change a system's computation in any way, but provide supplemental nonfunctional extensions[4]. Furthermore, we define a **plug-in** as a technical realization of an extension service. A plug-in defines a frame, which can be exploited to express arbitrary extension services. Consequently, we define a **proxy** as a facility that integrates extension services into existing systems by the management of a set of plug-ins.
In sections 7 and 8, we describe a framework that allows to transparently add extension services into EJB applications. Since, there exist no standardized interception mechanism for the platform, the framework is an application-level proxy-based solution. However, it exemplifies all aspects (complex execution order of services, conditional execution, dynamic reconfiguration) described in this paper. We outline the usage of the framework in section 8.

3 Dynamic Integration of Container Services

Changing requirements often make it necessary to provide new functionality, like logging or monitoring, for use by components in a running software system. Existing middleware standards do not allow for smooth integration of new

[4] The proxy framework proposed can also be seen as an 'extension service' that allows the integration of arbitrary services. However, we only use the term *extension service* for services that extend a system

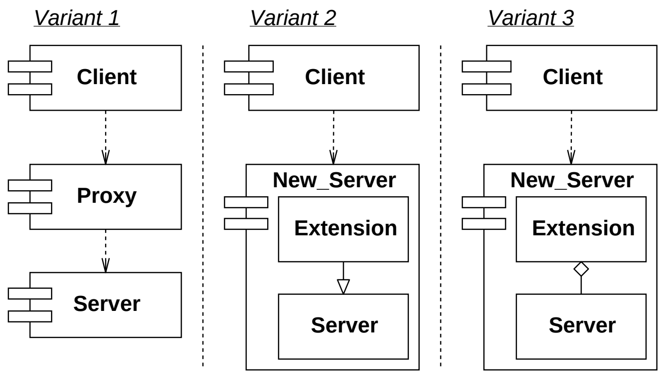

Fig. 1. Application Level Integration

services. Such services would have to be implemented as components. Existing components can only use those components explicitly, which makes it necessary to re-engineer the affected parts of the system. Common middleware standards avoid this problem by facilitating implicit use of standard services. For instance, in an EJB environment the container provides services to Enterprise JavaBeans as described in their deployment descriptor. Whenever the service requirements of an Enterprise Bean changes, it can be adapted by changing the contents of its deployment descriptor and redeploying it. However, the EJB specification only considers a limited number of standard services and provides no means for integration and implicit use of extension services. Java features such as custom class loaders cannot be used here because of the programming restrictions imposed by the EJB specification.
The framework described in this paper is to overcome this problem by facilitating dynamic integration of extension services and their implicit use by existing components without the need to redeploy them. As a result of this approach, the re-engineering effort for existing components associated with requirement changes concerning non-functional services will decrease, the availability of the system will improve, and third-party components will be able to use services for which they were not developed.

3.1 Pluggable Extensions

As described in section 2, each component has a proxy that intercepts method calls directed at the component and executes the relevant extension services. To facilitate dynamic configuration of the system by changing service properties, the static implementation of the interceptor has to be separated from the service code. That led to the introduction of a plug-in concept: the proxy merely represents an interface to the system extensions, which are implemented as exchangeable plug-ins. To execute the new functionality, the proxy reifies a method call after intercepting it and sends it to the plug-ins. In order to operate

properly, the plug-ins require information about the actual method call and additional context information, e.g. data about the interfaces of the encapsulated component.
In order to be able to receive reified method calls, a plug-in has to provide a generic method that takes information about a method call as arguments. Such information would include data about the called method, e.g. name and parameter types, and the arguments of the actual call. The structure of this method's implementation would be similar to an around advice in the aspect-oriented language AspectJ [15]. While the plug-in is processing the method call, it has the choice of forwarding it. The target of the forwarding step is another object that provides the generic method described above. In most cases this will be another plug-in. Eventually, the last plug-in has processed the reified call and forwards it. Now the requested method of the wrapped component has to be executed, which is done by the proxy shipping the method call to the component. For this purpose, the proxy has to provide such a generic method as well, and the proxy itself must be placed at the end of the call chain.
The forwarded method call does not necessarily need to be the same as the received one: the plug-in can change its arguments or even the information about the requested method itself, which would result in another method being called. After the call has returned, the plug-in can do further operations, including changes to the result value.
A number of context-aware services require further information, e.g. about the the plug-in's working environment or the identity of the client that has called a method of the encapsulated component. For that purpose each plug-in receives a special context object at some point in its lifecycle. By using this object in the implementation of the plug-in, it can be developed independent of a special runtime environment, which will increase its reusability. For instance, data about methods offered by the encapsulated component can be used during the forwarding step if another method is to be called. Instead of hard-coding the method call, reflective information provided by the context object should be used.
As stated above, system extensions offered by a plug-in must be implemented in a generic method. In some cases the additional functionality can completely be encapsulated in a single plug-in. However, some services are too complex, e.g. because they require data shared by several components or plug-ins. In those cases it could become necessary to use an external component that provides services used by several plug-ins.

3.2 Configuring the System at Runtime

As stated above, the described framework allows for configuration of the system at runtime. This means that new plug-ins can be integrated into the system and distributed among the proxies. The *proxy manager* serves that purpose. It is the central component that can be used to

- transfer a new proxy into the system,
- manage the configuration of the proxies, and

– distribute the plug-ins.

There are several stakeholders that want to use the proxy manager. After creating a new plug-in, developers have to make it available by sending it accompanied by a meaningful description to the proxy manager. Then the proxy manager assigns a unique id to the plug-in and stores it persistently. From that point on, the system administrator can use the new plug-in within proxy configurations. For these purposes the proxy manager offers methods for querying and updating proxy configurations. The configurations are also stored persistently to increase robustness. After changing the configuration of a proxy, relevant plug-ins are combined and sent to the concerned proxies.

4 Combining Plug-ins

The previous section described the general structure of a plug-in implementation and how method calls are forwarded. Before a number of plug-ins can be delivered to a proxy by the proxy manager, they have to be assembled in a single structure. The problem of functional overlaps between concerns has been addressed in the described framework by facilitating controlled execution of plug-ins.

4.1 Feature Interactions

Whereas separation of concerns during the early phases of software development has been thoroughly investigated, their combination in a software environment is still a major issue. In a perfect scenario, all concerns are disjoint, i.e. there are no functional overlaps. A single concern would be implemented as a single plug-in, and the execution of this plug-in would not affect other plug-ins. Other technologies, e.g. the composition filter model, require concerns (implemented as filters) to be disjoint [3].
Unfortunately, in many cases concerns do have overlaps in their functionality, and the execution of one concern interacts with the execution of other concerns. This is commonly known as the *feature interaction problem.* Before a software system can be assembled by combining 'features', possible interactions among those features have to be identified. As stated in [5], the effort of this process rises exponentially with the number of features. But not every feature interaction can be identified by statically examining the system, dynamic feature interactions only occur and therefore have to be analyzed at runtime.
When developing a combination mechanism for plug-ins, the possible occurence of feature interactions has to be considered. After identifying interactions between concerns, the developer must be given the opportunity to incorporate this knowledge into a structure which represents the combined functionality of several plug-ins, but is also flexible enough to handle feature interactions programmatically. A static list seems to be inappropriate, because the sequence of execution cannot be controlled except by rearranging the list. A single element of the list could change an incoming method call or 'ignore' it by simply forwarding

or dropping it. But if this mechanism were used to control feature interactions, changes to the method call would affect all subsequent plug-ins and thus induce further side effects. It would be impossible to restrain the changes to those plug-ins that the given plug-in interacts with. Moreover, it would also require the plug-in to be aware of its existence in a list and of interactions with other plug-ins. Because this knowledge is not part of the actual concern implemented in the plug-in, the plug-in itself would become more complex and less reusable.

4.2 Controlled Execution of Plug-ins

In the described framework extension services and their controlled combination must be developed separately. In order to improve reusability, a plug-in should contain the implementation of only a single extension service without references to other services. To manage interactions between several extension services, a special *controller plug-in* which governs their execution was introduced. Even though such a controller plug-in does not implement a generic concern, it has the same interface as the service-implementing plug-ins, i.e. it provides a generic method for receiving reified method calls. From the proxy's point of view, service-implementing and controller plug-ins are indistinguishable. The communication between the proxy and its plug-ins is defined by the specification of the generic method, which means that a proxy is independent of a certain configuration. Controller plug-ins are different. They are aware of their environment because they are specifically designed for a given context. A controller 'knows' the plug-ins it is working with and can use that knowledge, e.g. by calling special methods to check the state of a plug-in, or by forwarding the method call only to a subset of all its plug-ins.
Plug-ins are assembled in a tree-like structure with the proxy as its root and plug-ins as the other elements. After intercepting a method call, the proxy reifies that call and sends it to the plug-ins that are directly connected to the proxy. In the example shown in figure 2, the plug-ins P_1, P_2, and P_3 are used by the proxy. Plug-in P_1 controls plug-ins P_{11} and P_{12} and provides their combined services by forwarding the reified method call to them. There is no limit on the height of the plug-in structure, which means that controller plug-ins can use other controllers, too.
To simplify the development of controller plug-ins, an abstract super class has been developed. It exploits the same combination mechanism used by the proxy, i.e. it appends itself at the end of each used plug-in, as depicted in figure 2. When the current plug-in set is requested by a proxy, the proxy manager assembles the plug-in structure and makes it available for the proxy.

4.3 Plug-ins in Practice

Many kinds of feature interactions are conceivable, and with the described plug-in structure only a limited number of them can be handled. More complex structures are possible, but they are likely to require more effort to manage the proxies. In the implementation of a constraint checker based on this framework, the

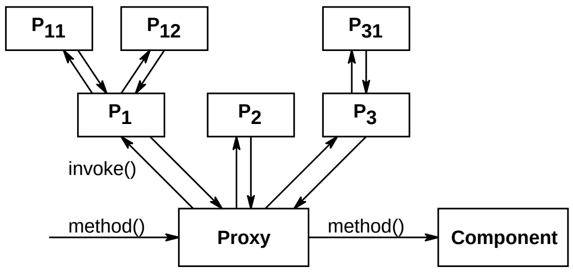

Fig. 2. Method calls in a plug-in configuration

tree structure has proven to be sufficient for most problems which result from semantical overlaps between two or more context-based constraints. Problems and restrictions of this framework are discussed below.

5 Providing Conditional Expressions

The proxy framework allows transparent integration of extension services into existing systems. Sometimes the execution of these services is restricted by constraints. We propose the separation of constraint expressions and the service's logic in the proxy framework, in order to isolate non-functional requirements. Thus, services are independent of changed requirements and the framework is able to optimize constraint evaluation.
The separation of these two aspects is localized in plug-ins, which are composed out of two parts. The functional part of the extension service is transparently encapsulated by a conditional part, which is separately specified. A plug-in is generated out of the conditional part and the extension service. New constraints result in a regeneration of plug-ins with new conditional parts. Thus, constraints can be freely exchanged without the need to modify the extension service.
A major problem of constraint evaluation at application-level approaches is based on the computation of required system properties. Normally, some system properties are not accessible at application level. For instance the caller ID property is not resolvable in EJB implementations.
In the following we first give an overview of the constraint evaluation concept and then introduce an implementation approach.

5.1 Concept of a Rule-based Evaluation

We propose to integrate conditional logic into a proxy concept based on Event Condition Action (ECA) rules. ECA rules are typically applied in active databases [27], where they are used to check for basic database operations and where additional actions are triggered when specified conditions are met [5].

[5] The notion of ECA rules has to be seen as analogous to ECA rules in database systems. However it does not fulfill exactly the same functionality.

ECA rules are exemplified in figure 3. The *event clause* specifies the event that must occur to trigger the ECA rule. An event can be related to a specific, or to several method calls. The *conditional clause* contains a expression which has to be true in order to start the action defined in the *action clause*.

```
declare rule rule_name
 on event
 if condition
 do action
end rule
```

Fig. 3. ECA Rule Definition

```
declare rule logging_example
 on invocation of the component 'SalesMngt'
 if call parameter 'age' > 17
 do log
end rule
```

Fig. 4. Logging Example Rule

Conditions have to be stated in a subset of first-order predicate logic. They can refer to communication parameter values, the the system state or to information provided by other components. System information is far more difficult to evaluate and needs supporting components, which gather actual system values. Actions represent the extension services to be integrated.

6 Implementation of the Framework

The concepts described in the previous sections have been implemented in the **EJBComplex** framework. A software system based on this framework consists of *com*ponents and *pl*uggable *ex*tensions. As the name suggests, the implementation is based on the EJB specification [8]. In order to ensure portability, the framework has been developed on application level, which means that any EJB container can be used.

6.1 The EJBComplex Framework

Similar to stubs, the proxy has to be generated before the Enterprise Bean can be deployed. In contrast to stub generation, the result of this step is not a vendor-specific artifact, but another Java class that can be used as the implementation class of a Bean. The related deployment descriptor will be adapted as well. Three variants of proxies on application level have been discussed in section 2. Variant 3 of figure 1 is the best solution because the proxy fully wraps the old Bean. In this case, the generated Bean class replaces the old implementation class, i.e. there is only a single component that represents the proxy and the old Bean. But this approach is only feasible for Session Beans, Message-Driven Beans, and Entity Beans with bean-managed persistence or CMP 1.1, because the old bean class has to be instantiated. This is not possible for Entity Beans with container-managed persistence 2.0, because their methods for accessing field values must be abstract. Because of this restriction, a separate Entity Bean with bean-managed persistence that holds a reference to the old Bean has to be generated, as shown as variant 1 in figure 1.

Proxies and plug-ins have one thing in common: they must provide a generic method for receiving reified method calls. The interface `MetaObject`, which must be implemented by all proxies and plug-ins, contains this method. Plug-ins must also offer methods that can be used by the proxy to combine plug-ins or transfer the context object. Because the implementation of those methods is likely to be similar in all service-implementing and controller plug-ins, the abstract super classes `Plugin` and `AbstractController`, respectively, have been developed. Plug-in developers only need to implement the generic method `invoke`.

The proxy manager is the interface between the developers or administrators and the EJBComplex framework. It is realized as a Session Bean, whose remote interface provides methods for configuring the proxies. Because of restrictions that are described below, it has a local interface that is used by proxies to obtain the current plug-in structure. Instances of plug-ins and proxy configurations are stored persistently in a database.

6.2 Problems and Restrictions

EJBComplex is implemented on application level, which has the advantage of better portability of the framework because it is based solely on the EJB specification and thus can be used with any EJB container. But there are also several disadvantages connected with this approach.

First of all, it is impossible to obtain control over standard container services. One of those services is *instance management*. The server controls a pool of instances of a certain Bean, and assigns an identity to an instance when required. This caused a major problem for the implementation of the plug-in distribution by the proxy manager. The best method would be to send plug-ins to the proxies of a certain Bean immediately after its configuration has changed. That is not feasible because the proxy manager cannot get references to all instances of a Bean, which would also include passivated instances. Instead, the proxy has to request plug-ins from the proxy manager each time one of its methods has been called. Optimization techniques have been used in the implementation to reduce communication costs and database access.

Certain types of services require the forwarding of context information. An example for such a service is *security management* as described in the EJB specification, because the identity of the caller (*principal*) must be transferred. Client-side proxies that enrich a method call with such context information are a solution to this problem. But without changing the stubs and skeletons generated by vendor-specific tools, this behaviour could not be achieved.

7 Application Scenario: Context-Aware Extension Services

7.1 Overview

The previous sections have discussed implementing extension services. This section focuses on configuring them. An extension service can enforce a requirement. One requirement can apply to many communication paths between several components. In order to enforce the requirement, all relevant plug-ins must be configured accordingly. The manual configuration of each individual plug-in at each communication path is expensive if many components are involved in the requirement or if the components involved change frequently. For example, several plug-ins may be needed at different places to enforce the following privacy policy: "*Components used in the workflow 'Create Report' must be logged if they invoke any component that handles personal data*". This section presents an approach that facilitates to determine which plug-in(s) must be deployed at which communication path(s). Its basic concepts can be explained in just a few sentences:

1. The components are annotated with formatted metadata called 'context properties'. A context property describes its component's context. In this case, 'context of a component' does *not* refer to 'required interfaces and the acceptable execution platforms' as defined in [26]. Instead, context is 'any information that can be used to characterize the situation of a component' as defined in [9].
2. Section 7.2 explains a new notion of constraints called CoCons. A Cocon can select the components involved in a requirement according to their *context property* values. CoCons facilitate configuring the EJBComplex framework as discussed in section 7.3.

7.2 Context-Based Constraints (CoCons)

The context of a component can be expressed as metadata. Metadata is typically defined as 'data about data'. According to [1], the attribute-value pair model is the commonly used format for defining metadata today. As well, we suggest expressing the context of a system component in the simple attribute-value syntax: a **context property** consists of a name and a set of values. The context property values associated with a component describe the component's context. Two examples for context properties are used in this paper:

Personal Data: Its values 'True' or 'False' signal whether the associated component handles data of private nature.
Workflow: Its value(s) name the current workflow(s) in which the associated component is involved.

The primary benefit of enriching components with context properties is to identify those components that are involved in a requirement. One requirement can affect several possibly unassociated components. A **context-based constraint** (CoCon) can indirectly select its constrained elements according to their context property values. As defined in [7], a CoCon relates *two sets* of elements and expresses a predicate for each *pair* of related elements. The **C**ontext-Based **C**onstraint **L**anguage **CCL** introduced in [6] consists of 21 different types of CoCons for defining requirements for component-based systems. This paper focuses on the family of communication CoCons because they can control extension services. A CoCon relates two sets of components. Let the component x be element of the one set, and the component y element of the other set. Only one of the communication CoCons is discussed here: a x `MUST (NOT | ONLY) BE LOGGED WHEN CALLING` y CoCon specifies that (ONLY) a communication call from x to y must (NOT) be logged. The elements (x or y) of each set can be selected via a context condition: if the context property values of a component comply with the context condition then the CoCon constrains this component: The syntax of CCL is not explained here, because it resembles plain English and is easily understood. For example, the privacy policy described in section 7.1 can be expressed in CCL as follows:

```
ALL COMPONENTS WHERE 'Workflow' = 'Create Report' MUST BE LOGGED WHEN
CALLING ALL COMPONENTS WHERE 'Personal Data' = 'True'.
```

CoCons can both directly and indirectly select elements. For example, the expression 'c_1, c_4 and c_7' *directly* selects three components. The same set of components can be selected via the context condition `ALL COMPONENTS WHERE 'Personal Data' = 'True'`. The indirectly selected components are anonymous. They are not directly named or associated, but indirectly described according to their context property values. The indirect selection *automatically* adapts to changed components or context changes, while the direct selection doesn't. For instance, eventually a new component will be managed by the system that was not managed yet when writing down the CoCon. The new component c_{31} is not selected by the direct selection given above. On the contrary, the indirect selection will automatically apply to c_{31} as soon as c_{31}'s context property 'Personal Data' has the value 'True'. The indirect selection statement must not be adapted if system components or their contexts change. Instead, the indirectly selected components are identified by evaluating the context condition each time when the system is checked for whether it complies with the CoCon. In order to enforce a `LOGGED WHEN CALLING` CoCon, a context-aware logging-service can control the communication call between the constrained components as described next.

7.3 Configuring the EJBComplex Framework via Communication CoCons

An extension service can be used to enforce a non-functional requirement, like the privacy policy discussed in the previous section. Many plug-ins at different communication paths can be needed in order to enforce one requirement. As explained in section 7.1, the manual configuration of each individual plug-in at each communication path is expensive if many components are involved in the requirement or if the involved components change frequently. If a requirement is expressed via a CoCon then all plug-ins needed to implement a corresponding extension service can automatically be configured as sketched next.

First, this section discusses how to automatically identify which plug-ins must be deployed at which places in order to enforce a CoCon. A CoCon identifies the constrained components via their context property values. The *range* of the allowed context property values must be defined for each component. For example, the context property 'workflow' might have the range (= allowed valued) 'Create Contract', 'Delete Contract' and 'Change Contract' for the ContractMgr component. Only a subset of the context property values defined in the component's context range can be associated with the component. For instance, the ContractMgr can be associated with the value 'Delete Contract' for its context property 'Workflow'. One CoCon can constrain several components that can invoke each other via different communication paths. In order to calculate which communication paths need a plug-in, the context property *range* of each component must be checked if it complies with the CoCon's context condition. A plug-In must be deployed at each communication path between each pair (x, y) of constrained components.

As explained in section 5, the functionality of a plug-in should be controlled by an ECA rule. One communication CoCon can be translated into *many* ECA rules as discussed in [16] and explained next. One CoCon can constrain many components. An ECA rule, on the other hand, refers to a particular communication path between two components. Thus, an ECA rule is generated for each communication path between all components that are constrained by the CoCon. In order to deploy all plug-ins needed to enforce a CoCon, the ECA rules have to be translated into an executable source code, have to be assigned to a corresponding extension service, and have to be deployed at appropriate proxies. In these ECA rules, the *event clause* specifies which communication path has to be monitored. The context conditions of the CoCon must checked in the *condition clause* of the ECA rule if they refer to context property values that can change at runtime. In this case, the context of the caller and the callee are checked at each call, and the plug-in is only triggered if caller context and callee context fulfil the CoCon's context conditions. Many notations for writing down metadata as attribute-value pairs exist and can be used for expressing the context of components. For example, the context of a component can be defined in its deployment descriptor in EJB systems. In Microsoft component frameworks , 'context properties' are already available. However, EJB deployment descriptors or Microsoft context properties are not supposed to change at runtime. Therefore, we suggest managing context properties via an extra component that provides their current values for each component. If a call fulfils the ECA rule's condition then an action is executed by the plug-in as stated in the *actions clause* of the ECA rule. This action must correspond to the Communication CoCon type in order to enforce the CoCon. For instance, the action 'log this call' corresponds to `LOGGED WHEN CALLING` CoCons.

The people who need a new requirement to be enforced often neither know the details of every part of the system nor do they have access to the complete source code. CoCons can be expressed via the language CCL similar to plain English. Moreover, CCL expressions stay on an abstract level that improves comprehensibility. When configuring extension services via CCL expressions, developers or administrator don't have to understand every detail of the system ('glass box view'). Instead, they only must understand in which context the components reside that must meet their goal. The context property values of a component can be adapted whenever the context of a component changes without changing the component or the extension service itself. Moreover, a component that was unknown when specifying the CoCon becomes automatically constrained by the CoCon simply by having the matching context property value(s). It can be unknown which components are involved in the requirement when expressing it in CCL. Hence, the constrained components can change without modifying the CoCon expression. The indirect selection of constrained components is particularly helpful in highly dynamic or complex systems. Every new, changed or removed component is automatically constrained by a CoCon according to the component's context property values. A flexible framework is needed in order to enforce Communication CoCons at runtime, because both contexts and requirements can change at runtime. Such a framework needs all the features provided by the EJBComplex framework: it needs complex execution order of plug-ins, conditional execution of plug-ins, as well as dynamic configuration of plug-ins.

8 Related Work

This paper presents a concept for integrating additional functionalities into existing systems without modifying the system's components. For this purpose, a mechanism is needed which is able to transparently integrate these functionalities into existing systems. Connectors, which are considered first-class entities in the field of software architecture [18,23,24], describe such a mechanism on an abstract level. Within existing middleware technologies, there are different ways to realize a connector [13,17,2]. However, this paper focuses on approaches using the proxy pattern [11], which allows the transparent integration of non-functional services [22,4,12,21].

The lack of decomposition features in many programming paradigms has been addressed by a number of research groups. Aspect-oriented programming [14] separates components, which represent the functional part of a system, from aspects, which cross-cut the system's functionality. The composition filter model [3] adds an additional layer of filters to extend the functionality of a base object. Our plug-in mechanism is closely related to these technologies, although we allow dynamic reconfiguration and use a more complex combination mechanism.

A CoCon defines a *network policy* between clients using network resources and the network elements that provide those resources. The Internet Engineering Task Force (IETF) has defined a policy model for policy-based networking in [19] An underlying assumption of this model is that policies are stored in a centralized repository. The policy repository is one of three important entities of the model. The other entities are the policy enforcement points (PEPs) and policy decision point (PDP). On the contrary, this paper suggests a decentralized approach for enforcing network policies. The EJBComplex framework provides a distributed architecture in which each plug-in (or PEP) stores a part of the policy as an ECA rule. Thus, EJBComplex plug-ins do not need to request a decision from a PDP server. The main interest in network policies is managing and controlling the *quality of service* (QoS). In contrast to prevailing network policy approaches, this paper does not focuses on QoS. On the contrary, the key notion of extension services is adding non-functional requirements to a system, like logging certain calls or encryption.One of the recent network policy approaches is the *path-based policy language* (PPL) described in [25]. It provides control over the traffic in a network by constraining the path the traffic must take. A PPL path may include wild card characters. The use of a wild card character is similar to using a context condition - both approaches can express one statement that constrains many paths. They differ, however, in the criteria for selecting the constrained paths. While PPL expressions use technical criteria for selecting the constrained paths, CoCons use semantical criteria. We believe that semantical criteria are more comprehensible, because the expression 'all nodes whose IP address starts with 123.45.67.*' does not tell *why* these nodes are selected, while 'all nodes which manage personal data' does.

9 Conclusions

Requirement changes demand continuous adaptation of a software system throughout its lifetime. This paper is concerned with non-functional aspects of a system, which can be transparently integrated using an interception mechanism without modifying existing components. Thus, this paper proposes the separation of non-functional requirements from system components. There are several existing approaches that engage the technical aspects of service integration. The objective of this paper is to enrich these approaches with additional aspects dealing with complex service order and conditional service execution.
The proposed framework demonstrates these objectives. Extension services are implemented as plug-ins, which can be dynamically integrated into the system without changing any of its components. The extension services are used implicitly by existing components. This is achieved by intercepting method calls directed at the component by the proxy and executing the plug-ins using a generic interface.
The interface to the framework is the proxy manager. It can be used to transfer plug-ins into the system and to change the configuration of proxies at runtime. The proxy manager is also responsible for combining the plug-ins designed for a certain proxy. For that purpose, a flexible tree structure is used. The configuration of a proxy may contain special plug-ins that control the execution of other plug-ins. Problems resulting from functional overlaps can be programmatically handled using this feature. The combination structure for plug-ins has proven to be sufficient for a number of semantic overlaps between plug-ins. Furthermore, the separation of conditional logic and functionality of an extension service is

proposed. Non-Functional Requirements are formulated as ECA rules. Based on these rules, plug-ins can be generated.
Components constrained by a *context-aware* extension service can be indirectly selected according to their context property values. A context-aware extension service automatically adapts whenever the context of a component or a user changes without changing the component or the extension service itself. Hence, context-aware extension service easily adapts to changed contexts, changed requirements or changed configuration. Communication CoCons can define context-aware extension services. However, contexts and requirements can change at runtime. Hence, all features of the proposed framework are needed in order to transparently realize context-aware extension services: complex execution order of plug-ins is needed as well as conditional execution of plug-ins as well as dynamic configuration of plug-ins. The EJBComplex framework has prototypically been implemented providing these features. It has successfully been used for monitoring CoCons at runtime.

References

1. Thomas Baker. A grammar of dublin core. *D-Lib Magazine*, 6(10):47–60, October 2000.
2. BEA Systems, Inc. *BEA WebLogic Server, Using WebLogic Server Clusters*, March 2001.
3. L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51 – 57, October 2001.
4. Marko Boger, Toby Baier, Frank Wienberg, and Winfried Lamersdorf. Structuring QoS-supporting services with smart proxies. In *Extreme Programming and Flexible Processes in Software Engineering - XP2000*. Addison-Wesley, 2000.
5. T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin. The feature interaction problem in telecommunication systems. In *7th International Conference on Software Engieering For Telecommunication Switching Systems*, pages 59 – 62, 1998.
6. Felix Bübl. The context-based constraint language CCL for components. Technical Report 2002-20, Technical University Berlin, available at www.CoCons.org, October 2002.
7. Felix Bübl. Introducing context-based constraints. In Herbert Weber and Ralf-Detlef Kutsche, editors, *Fundamental Approaches to Software Engineering (FASE '02), Grenoble, France*, volume 2306 of *LNCS*, pages 249–263, Berlin, April 2002. Springer.
8. Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise javabeans specification, version 2.0. Sun Microsystems, `http://java.sun.com/j2ee`, August 2001. Sun Microsystems.
9. Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.
10. Hans Rohnert Douglas Schmidt, Michael Stal and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Distributed Objects.* John Wiley & Sons, 2000.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

12. Rushikesh K. Joshi. Modeling with filter objects in distributed systems. In *Proceedings of the 2nd Workshop on Engineering Distributed Objects, (EDO 2000)*, volume 1999 of *LNCS*, pages 182 – 187. Springer Verlag, November 2000.
13. Yariv Kaplan. API spying techniques for windows 9x, NT and 2000. http://www.internals.com/articles/apispy/apispy.htm, 2001.
14. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer, New York, 1997.
15. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59 – 65, October 2001.
16. Andreas Leicher and Felix Bübl. External requirements validation for component-based systems. In A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Ozsu, editors, 14^{th} *Conference on Advanced Information Systems Engineering (CAiSE '02), Toronto, Canada*, volume LNCS 2348, pages 404 – 419, Berlin, May 2002. Springer.
17. Dmitri Leman. Spying on com objects. *Windows Developer's Journal*, July 1999.
18. Nikunj R. Mehta. Software connectors: A taxonomy approach. In *Workshop on Evaluating Software Architectural Solutions 2000.* Institute for Software Research University of California, Irvine, 2000. `http://www.isr.uci.edu/events/wesas2000/position-papers/mehta.pdf`.
19. Bob Moore, Ed Ellesson, John Strassner, and Andrea Westerinen. Policy core information model - version 1 specification (rfc 3060). Technical report, The Internet Society, 2001.
20. JBoss Organization. Jboss website. http://www.jboss.org, December 2001.
21. G. S. Reddy and R. K. Joshi. Filter objects for distributed object systems. *Journal of Object Oriented Programming*, 13(9):12–17, January 2001.
22. E. F. Robert, S. Barret, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Communications of the ACM*, 45(1):116–122, January 2002.
23. Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *ICSE Workshop on Studies of Software Design*, pages 17–32, 1993.
24. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. PH, April 1996. ISBN 0131829572.
25. Gary N. Stone, Bert Lundy, and Geoffrey Xie. Network policy languages: A survey and a new approach. *IEEE Network*, 15(1):10–21, January 2001.
26. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1999. ISBN: 0-201-17888-5.
27. Jennifer Widom and Umeshwar Dayal. *A Guide To Active Databases.* Morgan-Kaufmann, 1993.