An MDA Approach for Adaptable Components

Steffen Göbel

Institute for System Architecture
Dresden University of Technology, Germany
goebel@rn.inf.tu-dresden.de

Abstract. Components should provide maximum flexibility to increase their reusability in different applications and to work under changing environment conditions as part of a single application. Thus, adaptation and reconfiguration mechanisms of single components and component assemblies play a crucial role in achieving this goal. In this paper we present a model of Adaptable Components that allows modelling of adaptation and reconfiguration operations taking place at development, deployment or runtime. The concept of composite components is utilized to encapsulate adaptation operators and to map component parameters to different predefined internal configurations of subcomponents. The component model is not tied to a particular component platform. Instead, it can be mapped to existing component platforms like EJB using an MDA approach. Different Platform-Specific Models for the same target component platform enable tailored flexibility for particular component deployments. For example, a model can support or not support runtime reconfiguration. Extensions to UML diagrams are introduced to graphically model reconfiguration operations.

1 Introduction

Two of the key success factors of component-based software engineering are *reuse* of components in different applications and *composition* of complex applications out of well-defined and loosely coupled parts. Component platform standards like JavaBeans, Enterprise JavaBeans (EJB), Microsoft COM, and Microsoft .NET have crucially contributed to this success. To increase reusability and flexibility, components should support mechanisms to adapt them to different environments and to add new functionality.

This adaptation process can be carried out at different times in the component life cycle: at development time, at deployment time, or at runtime. However, components look quite differently and consist of different artefacts at these three times. For example, at development time a component might comprise a set of Java source files and an XML descriptor. At runtime the same component might consist of binary program code structured in classes and several data structures. As a result, the conceivable adaptation mechanisms also differ at the three times. Some mechanisms might change the source code of components and others might employ interceptors at runtime.

A variety of adaptation mechanisms has been developed by the research community and software industry, but most of them are only applicable at one time, either development, deployment, or runtime. A unified model describing adaptable components at all three times is missing. In this paper we introduce such a model for the development of Adaptable Components (ACs). It employs the concept of composite components to encapsulate adaptation mechanisms. *Adaptation* is considered as a superset of the concepts *reconfiguration*, *customization*, and *parameterization* in this paper.

The model of ACs combines the following ideas: (i) The adaptivity is expressed by *component parameters*. The notion of component parameters comprises component properties used, for example, in JavaBeans and type parameter used in C++ templates or Java generics. In a previous work [7] we also showed that QoS properties are also a kind of parameters. (ii) An AC as a composite component encapsulates a set of predefined internal configurations of subcomponents. Component parameters are mapped to these configurations. That means that the internal configuration of an AC changes if component parameters are changed. (iii) The model of ACs is platform and programming language independent. An MDA approach enables transforming of the model to different target component platforms. Currently, only Java is supported as programming language. (iv) Existing adaptation mechanisms can be integrated by special modelling operators.

The paper is structured as follows: In the next section we introduce our model of ACs, explain model constituents, and modelling techniques. In the third section the model is illustrated based on a simple example—a crypto component. The fourth section deals with the implementation of the model and, in particular, with the MDA approach to transform the model to different component platforms. The paper closes with an examination of related work, a conclusion, and an outlook to future work.

2 Model of Adaptable Components

This section first describes the constituents and concepts of our model of ACs. The concept of composite components is employed to encapsulate model constituents, especially subcomponents. The advantage of this approach is that ACs do not look and behave differently than other components from outside. Many constituents (e. g., adaptation operators and glue code) in the model are optional, which increases flexibility and enables several application scenarios.

The remaining part of this section deals with mapping of component parameters and modelling of configurations and reconfigurations by using UML diagrams. New UML stereotypes are introduced to describe structural reconfiguration operations.

2.1 Component Constituents

The model of ACs is schematically depicted in Fig. 1. ACs define explicit dependencies to other components by fixed sets of typed *provided* and *required* ports

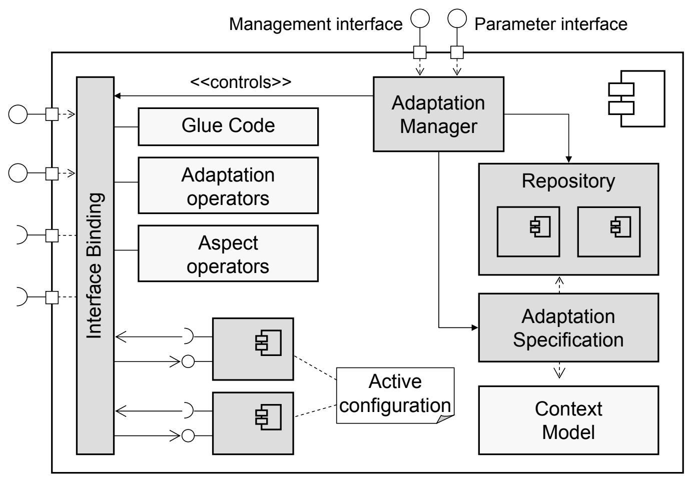


Fig. 1. Model of a adaptable component

each implementing a certain interface. Any communication with other components requires explicitly defined connections to these components via ports. The ports of an AC stay the same, even if the internal configuration is changed in the course of a reconfiguration. This means that the adaptation process stays transparent to other components of an application. A *management interface* (similar to the *equivalent interface* in CORBA Components) is used to navigate to required and provided interfaces and access the component parameter interface. The *parameter interface* provides read and write access to component value parameters (properties) by `get` and `set` methods.

We define the *configuration* of an AC as a set of subcomponents, a set of adaptation and aspect operators applied to subcomponents, a set of connections between ports of these subcomponents, and a binding between the external ports of the AC and ports of subcomponents. Thus, the *active configuration* determines the current runtime properties of a component, a kind of operating range. The set of valid configurations is defined by the *adaptation specification* containing the initial configuration and transitions to other valid configurations (see Sec. 2.2 for details).

The *component repository* contains all available subcomponents of an AC. A subset or all of these subcomponents, which are designated as *active subcomponents*, form an active internal configuration at runtime. Only active subcomponents process incoming method calls and the active configuration determines communication links. The *management interface* enables adding of new subcomponents and accordantly configurations at runtime. The *interface binding* as the central hub forwards requests of external ports to ports of active subcomponents and also connects internal subcomponents. The *adaptation manager* controls the interface binding and the reconfigurations using the adaptation specification. It thereby enforces consistency constraints and synchronizes reconfiguration operations.

Glue code is a special kind of subcomponent. The difference is that subcomponents are usually reused or at least intended to be reused by other applications whereas glue code is tailored to a particular AC. Glue code enables adding missing functionality to an AC (e. g., a new interaction protocol) that otherwise prevents subcomponents from working together.

Adaptation operators take a single subcomponent as input and transform it to a new subcomponent with a possibly different set of ports. They are typically applied to adapt incompatible port interfaces. Incompatibilities occur because component originally developed by different developers and for different applications are reused in a new application. For example, incompatibilities are caused by slightly different method names, different method signatures, and missing methods. Potential implementation approaches for adaptation operators include, but are not limited to:

– adapters deployed at runtime that extend or change some functionality of a subcomponent with or without providing a new interface;
– byte code transformers [9] that adapt subcomponents at deployment or loading time and before the application is started;
– source code transformers [2] that modify the source code of subcomponents before or during the compilation.

The actual implementation strategy is left open by our model.

Aspect operators are applied to a set of subcomponents and influence the behaviour at defined joinpoints like in aspect-oriented programming [10]. As with adaptation operators, different types of aspect operators can be employed at development, deployment, and runtime. Joinpoints include, but are not limited to, places before and after methods, component instantiations, component destructions, and reconfiguration operations.

So far we explained that the management interface of AC is used to change parameters and therewith the active internal configuration. However, these parameter changes need not necessarily be triggered by an external entity but can also be triggered by the AC itself, for example, if environment conditions change. The adaptation manager can utilize a *context model* of the environment to check conditions defined by the adaptation specification and change component parameters if specified. This way self-adaptable components can be modelled utilizing the mechanism of parameter mapping.

2.2 Modelling of Configurations and Configuration Variations

The modelling process of ACs defines the *adaptation specification* as described in the last subsection and thus all valid configurations of an AC. Two different approaches support the graphical specification of valid configurations: *complete configurations* and *configuration variations.*

Complete configuration—as the name suggests—graphically describe all subcomponents, connections, interface bindings, and parameter settings of a particular configuration by an UML component diagram. UML 2.0 supports all required concepts including the description of interface bindings using the delegate construct from the composite structure package. The description of complete configurations for ACs is useful if only a small set of different configurations exists and if these configurations differ in many aspects (e. g. different subcomponents and connections).

However, modelling of complete configurations is not very flexible if several configurations of an AC have only a few structural differences. For example, it should be possible to model an AC where an integer parameter specifies the number of instances of a particular subcomponent. Instead of modelling many complete configurations with different numbers of this subcomponent, it is sufficient to model the initial configuration with one subcomponent and the *reconfiguration operations* that add one subcomponent. Applying these reconfiguration operations multiple times to the initial configuration can create all desired configurations.

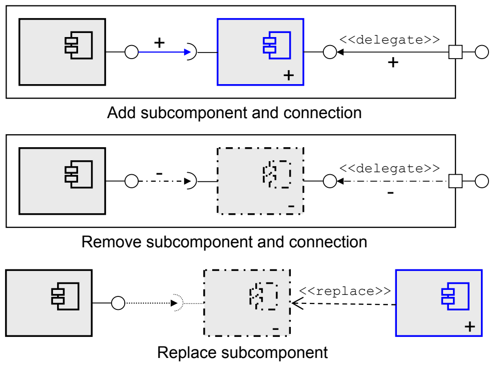
Thus, configuration variations as the second model approach describe changes applied to a particular initial configuration to indirectly define a new configuration. Six different atomic reconfiguration operations can be identified to describe a transformation from one configuration to another one:

– changing component parameter
– adding a connection between subcomponents or between an external interface and a subcomponent's interface
– removing a connection between subcomponents or between an external interface and a subcomponent's interface
– adding a subcomponent
– removing a subcomponent
– replacing a subcomponent by another subcomponent

It can easily be proven that these reconfiguration operations are sufficient to transfer any configuration to any other configuration[1].

UML has no built-in mechanisms to model reconfigurations of component nets, but the UML meta-model can be extended using stereotypes or new metaclasses for new concepts. Thus, we defined extensions for each of the above mentioned reconfiguration operations. New connections and subcomponents are highlighted and mark with a plus symbol as stereotype. Removed connections and components are drawn with a dashed line and a minus symbol as stereotype. Fig. 2 depicts the graphical notation of the extensions. These extensions can now be utilized to graphically model configuration variation consisting of several reconfiguration operations with UML diagrams. The developer takes a component diagram of a particular configuration as starting point and then graphically models the reconfiguration operations to get to the new configuration.

[1] A trivial transition first removes all connections and subcomponents of the old configuration and adds all new subcomponents and connections later on.



+
<<delegate>>
+
+
Add subcomponent and connection
-
<<delegate>>
-
Remove subcomponent and connection
<<replace>>
+
Replace subcomponent

Fig. 2. UML extensions for reconfiguration modelling

2.3 Mapping of Component Parameters

In general, the parameter mapping is defined by a function with component parameters as arguments that selects a predefined configuration and an ordered set of configuration variations applied to this configuration. Additionally, a parameter can influence parameters of subcomponents using the *change parameter* reconfiguration operation.

The setting of component parameters, which then selects the active configuration, is considered at four different times (see also Sec. 4.3):

Development time The developer can optionally define a default value for a parameter. It can be overridden at deployment time or runtime.

Deployment time The parameter value is set during the deployment of the application.

Creation time The parameter value is set during creation of a new component instance.

Runtime The parameter value is changed at an already existing component instance during the runtime of the application. This process is generally called *reconfiguration*.

Development, deployment, and creation time parameter settings can be supported quite easily. The components just need to be initialized and wired according to the resulting configuration. However, changing parameters at runtime, which possibly leads to a reconfiguration of components, requires considerably more effort. Particularly, the runtime environment of components has to address the following issues that can also be found in database transactions: *atomicity* – a reconfiguration must be executed completely or not at all; *consistency* – an AC must be in a valid state before and after a reconfiguration transition; and *isolation* – a reconfiguration must be executed without the interference of other reconfigurations or application activities.

Two kinds of parameters are considered: parameters with a finite set of possible values (*enumeration*) and parameters with integer or real values in a certain interval. In the following, we describe a few special forms of parameter mapping.

- The simplest form of parameter mapping assigns a particular configuration to each enumeration parameter value. For example, a component parameter “compressionAlgorithm” with the possible values “bzip2” and “gzip” could be mapped to two different configurations, respectively.
- A configuration is assigned to an interval of integer or real parameters. For example, a component parameter “compressionLevel” with a allowed range from 0 to 10 could be mapped to two different configurations in the intervals 0–5 and 5–10.
- Parameters are directly or indirectly mapped to parameters of subcomponents.
- An integer parameter specifies how often a configuration variation should be applied to a configuration.

3 Example: Crypto Component

In this section a crypto component is taken as a simple example to illustrate the use of ACs. The crypto component shall support a set of symmetric encryption algorithms (AES, Twofish, Blowfish). The interface contains only two methods: `encrypt` and `decrypt`. They enable encrypting and decrypting of given byte arrays with a given encryption key. Different loss-less compression algorithms (deflate, bzip2) are also part of the crypto component to optionally compress data before encryption and decompress it after decryption, respectively. Data compression is useful to reduce data size and to maximize the entropy of data, which complicates possible attacks on encryption algorithm.

Several parameters are exposed to adapt the crypto component:

cryptoAlgorithm is an enumeration parameter to select the active encryption/decryption algorithm (values: AES, TWOFISH, BLOWFISH).

key is an byte array parameter to set the active encryption key. This parameter does not influence the configuration of the crypto component.

compressionAlgorithm is an enumeration parameter to select the active data compression algorithm (values: DEFLATE, BZIP2, NONE).

If a user of the crypto component changes these parameters, a different internal configuration of the crypto component as depicted in Fig. 3 is selected.

The crypto component consists of three subcomponents for encryption algorithms and two subcomponents for compression algorithms. We assume that these subcomponents are available as libraries (e. g., open source projects) and need not to be implemented. Glue code implements the coupling of encryption/decryption and compression/decompression because this functionality was not supported by the crypto algorithms. All it does is calling the compression method of a compression subcomponent before calling the encryption method of a encryption subcomponent and accordingly for decompression and decryption. An adaptation operator is required for the Twofish subcomponent because it has a slightly different and therefore incompatible interface than the other encryption components. The methods for encrypting and decrypting are named

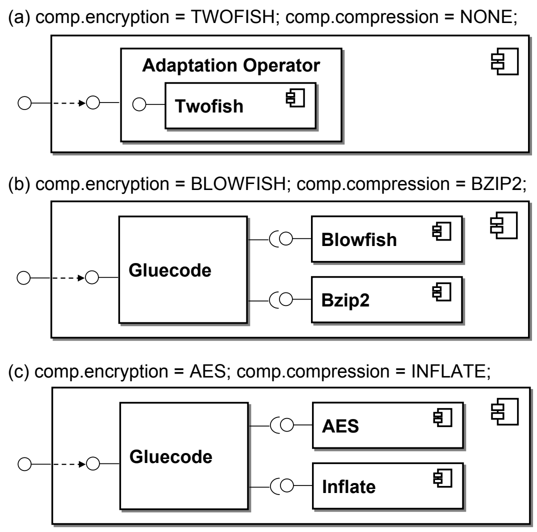

(a) comp.encryption = TWOFISH; comp.compression = NONE;
Adaptation Operator
Twofish
(b) comp.encryption = BLOWFISH; comp.compression = BZIP2;
Gluecode
Blowfish
Bzip2
(c) comp.encryption = AES; comp.compression = INFLATE;
Gluecode
AES
Inflate

Fig. 3. Example: Crypto component with different internal configurations

`encryptBytes` and `decryptBytes`. The adaptation operator internally maps the different methods pairs to change the interface to be compatible with the other subcomponents.

The crypto component is intended to be reused in different applications. For example, the following use cases are conceivable:

– The crypto component is used in a application with a fixed configuration, for example, always with AES and Inflate. This means that all parameters except the encryption key are set at development or deployment time.
– The parameters of the crypto component are set when creating new instances. However, once the instances have been created, parameters are immutable.
– The parameters of the crypto component are changeable at any time to give maximum flexibility.

Each of these scenarios enables different optimizations (see Sec. 4.3) but this should be completely transparent to the application developer. For example, in the first scenario unused subcomponents can be omitted whereas the last scenario requires additional program code to handle runtime reconfiguration.

4 Implementation and Mapping to Component Platforms

Our model of ACs presented in Sec. 2 is independent of any component platform and does not define any implementation details deliberately. However, the model is useless without an actual implementation. There are at least two implementation strategies: (i) all model constituents and concepts are directly supported by the component platform and associated framework or (ii) all model constituents are mapped to an existing component platform (e. g., EJB or JavaBeans) and missing functionality is either generated or implemented as auxiliary services. In this paper we only describe the latter approach but in a previous work [8] we also showed the feasibility of the first approach. The main advantage of the mapping approach is that existing application servers and component platforms can be reused. It is also feasible to integrate a single new AC in an existing component-based application with reasonable overhead. In this way expensive developments from scratch can be avoided.

In this section we first describe the implementation of the metamodel of ACs and a generic framework to support the life-cycle of ACs, especially creating of instances, parameter mapping, and runtime reconfiguration. Next, we show how the model can be mapped to a component platform in general, and to EJB in particular. We explain how the different model concepts can be emulated by the target component platform and which constraints are necessary for the development of components. Other component platforms might be supported in a similar way.

4.1 Metamodel of ACs

The metamodel of an AC contains all the information necessary to control AC's life cycle—from creation, over reconfiguration until destruction. It describes all subcomponents, adaptation operators, interfaces, configurations, variations, etc. We chose the Eclipse Modeling Framework (EMF, [6]) as tool for implementing the metamodel. EMF provides a lot of useful features that simplified our implementation. It employs a subset of MOF 2.0 called Ecore to store metamodels, it generates code to access instances of the metamodel (in our case AC models), it generates a simple Eclipse Plugin as editor for the metamodel, and it includes default support for persistence of model instances using XMI.

Fig. 4 depicts a simplified class diagram of AC's metamodel. `AdaptableComponent` is the entry point to this diagram. An AC has a set of `Configuration`s that are mapped to certain parameter values or value ranges via `ParameterMapping`s. A `Configuration` defines the set of `ComponentInstances` that are wired using several `Connection`s between component `Ports`. A `Reconfiguration` defines necessary `ReconfigurationOperation`s to switch from one configuration to another one.

The metamodel is not only used by runtime support of ACs (see next section) but also by the modelling tool to design and develop ACs. This modelling tool is currently under development and will be realized as an Eclipse Plugin.

4.2 Generic Life-Cycle Support for ACs

Although the mapping to different component platforms differs in many details, we have developed a generic framework that can be used and extended by platform-specific implementations. It contains a set of Java classes implementing

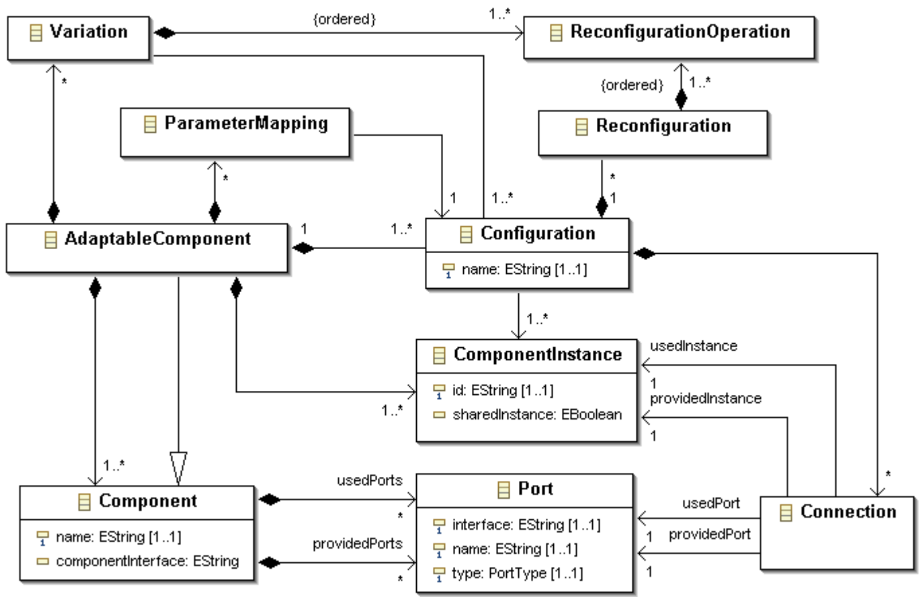


Fig. 4. UML class diagram of AC's metamodel

common functionality for managing AC's life cycle and relies on the implementation of the metamodel. It is employed to develop platform-specific adaptation managers, component repositories, and interface bindings (see Fig. 1). The offered functionality can be divided into three categories: creation of instances, runtime reconfiguration, and helper functions.

Creation of Instances Several steps are necessary to create an AC instance. First, the current values of component parameters are used to determine the active configuration of the AC. Next, all subcomponents including glue code and adaptation operators are instantiated according to the configuration. In the last step, ports of subcomponents are connected among each other and with external ports of the AC. These steps are not necessarily all performed at runtime of the AC. For example, the first step might also be performed during deployment of an AC and could generate appropriate code. The actual implementation strategy is defined by the platform-specific model (see Sec. 4.3).

Runtime Reconfiguration Parameter changes at runtime can trigger reconfigurations of ACs. A reconfiguration must adhere to the characteristics described in Sec. 2.2 and is performed in several steps. First, the new configuration is determined by the parameter mapping. Next, all component instances that are influenced by reconfiguration operations (e. g., an instance might be removed) must be stopped putting the component in an inactive state. Inactive means that no method call is currently in progress in this instance and new calls are blocked. Special caution is required to prevent deadlocks in this step. Next, all necessary reconfiguration operations are executed. In the last step, all blocks are removed and method calls can be continued.

Helper function Several functions (e. g., creating and removing connections and subcomponents) are provided to solve small tasks both during the creation of instances and the runtime reconfiguration. Many of them are intended to be implemented or extended by platform-specific code. Thus, they are hooks for platform-specific behaviour.

4.3 Model Mapping

The mapping of our model of ACs to a particular component platform requires that all model constituents and concepts described in Sec. 2 are supported or emulated by means of the target platform. The most important concepts to be mapped are composite components, explicit dependencies by ports, and component parameters. This leads to a Platform-Specific Model (PSM) and a platform-specific implementation, later on.

The emulation of unsupported features by the target component platform often requires defining constraints. That means that certain feature of the component platform must not be employed for developing ACs. For example, many component platform use name servers to acquire references to collaborating components and the accordant program code is tangled in the business logic. Using this feature could violate explicitly defined dependencies between components.

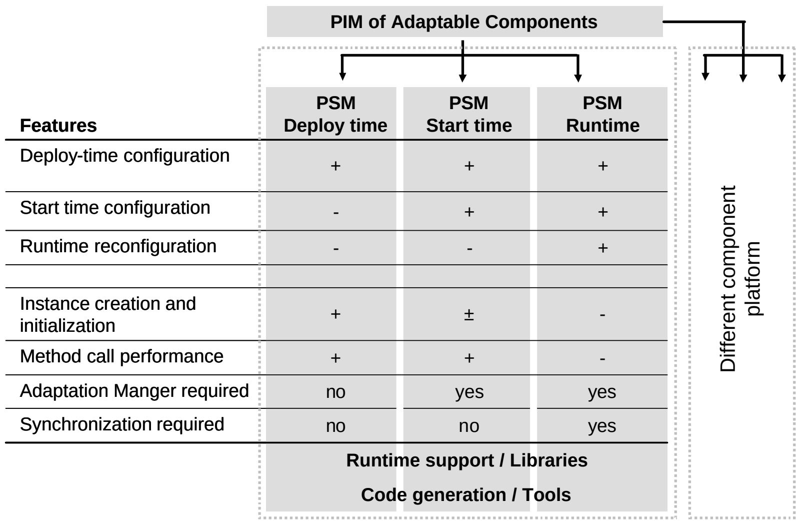
We consider at least three different PSMs for each component platform. The different features are illustrated in Fig. 5. The right level of flexibility for a certain AC can be chosen depending on application needs. This choice requires no changes to the AC by the developer. It is all transparently managed by the runtime framework and development tools.

4.4 Example: Model Mapping to EJB

We chose EJB as target component platform to demonstrate how a model mapping can be achieved. As described in the previous section, each model constituent and concept must be emulated by features of EJB. Due to limited space we only focus on the "PSM Runtime“ according to Fig. 5. Since runtime reconfiguration is the most complicated part, the other two PSMs are even simpler to realize.

EJB does not directly support composite components, which are a key concept in our component model to encapsulate all other constituents. However, it is feasible to flatten ACs and emulate this concept with functionality available in EJB. We defined the following mapping rules and constraints:

– A single stateful session bean is generated for every AC and its home interface is bound to JNDI (Java Naming and Directory Interface). This session bean also implements the management and parameter interface. Method calls are forwarded to the adaptation manager.


PIM of Adaptable Components
Different component platform
Runtime support / Libraries
Code generation / Tools

Features	PSM Deploy time	PSM Start time	PSM Runtime
Deploy-time configuration	+	+	+
Start time configuration	-	+	+
Runtime reconfiguration	-	-	+
Instance creation and initialization	+	±	-
Method call performance	+	+	-
Adaptation Manger required	no	yes	yes
Synchronization required	no	no	yes

Fig. 5. Overview and comparison of the model mapping

- Every subcomponent is bound to JNDI with unique names only known to the AC session bean.
- A `get`*PortName* method is generated for every subcomponent. It hides the process of getting references by JNDI from the component logic. The Adaptation Manager initializes all references during start-up or after a reconfiguration.
- Business methods of subcomponents must not directly use JNDI but the generated helper methods to access other components. This constraint is necessary to enable explicitly defined connection between components and reconfiguration at runtime.
- Glue code is encapsulated in a session bean and otherwise handled like any other subcomponent.

Other components outside the AC only work with the generated session bean like with any other session bean and need not to be aware of implementation details.

The support of runtime reconfiguration of ACs with EJB is relatively easy to implement. The EJB specification enforces non-reentrant instances. The EJB container must ensure that only one thread can be executing an instance at any time. That means that a method call and a parameter change possibly triggering a reconfiguration can never happen at the same time. Thus, no additional code needs to be injected to block method calls during a reconfiguration.

5 Related Work

Many Architecture Description Languages (ADL, [12]) use composite component concepts to provide a uniform view of applications at different abstraction levels. However, ADLs focus on highly distributed systems and many view configurations statically. Exceptions are Darwin [5] and Rapide [11] supporting constrained changes of configurations as well as C2 [14] and Wright [1] supporting almost arbitrary configuration changes at runtime. As opposite to our approach, reconfigurations are specified by program code or scripts, a model transformation to different component platforms is not considered, and a mapping of component parameters to different configurations is not supported.

OpenORB and OpenCOM [4, 3] aim at developing a configurable and reconfigurable component-based middleware platform. They support composite components based on enhancements of Microsoft COM and, especially, offers extensive meta-programming interfaces to adapt components and the middleware itself at runtime. ObjectWeb Fractal [13] completely focuses on the development of an adaptable composite component model with reflection and introspection capabilities to monitor a running system. Components need not to adhere to a fixed specification, but can rather choose from an extensible set of concepts and corresponding APIs depending on what they can or want to offer to other components. However, both approaches focus on the definition of appropriate interfaces and not on the design of adaptable components. Reconfigurations must be developed by program code using meta-programming APIs. The reconfiguration mechanisms are integrated in a special-purpose component platform.

6 Conclusions and Outlook

In this paper we presented a model of adaptable components that is independent of a particular component platform. It enables mapping of component parameters to different internal configurations and, therewith, encapsulating of structural adaptation. Adaptation operators and glue code as optional model constituents help to integrate and reuse existing components in new ACs. Different PSMs for a single target component platform enable tailored flexibility according to application needs.

We showed how this model can be mapped to the EJB platform as an example. Additionally, we have already implemented support for JavaBeans. Mappings to WebServices and JMX are currently under development. We also work on sophisticated tool support for designing and developing ACs.

Some detail problems will be addressed by further research: In some cases, the state of components must be transferred to replacement components in the course of a reconfiguration. This will be supported by special state transform operators. We will also investigate how the generated code, especially to synchronize reconfigurations, can be optimized.

References

1. R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, 1998.
2. U. Aßmann. *Invasive Software Composition.* Springer-Verlag, 2003.
3. G. S. Blair, G. Coulson, L. Blair, H. A. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, Self-Awareness and Self-Healing in OpenORB. In *Workshop on Self-Healing Systems (WOSS '02)*, pages 9–14, Charleston, SC, USA, 2002.
4. G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.
5. S. Crane, N. Dulay, J. Kramer, J. Magee, M. Sloman, and K. Twidle. Configuration management for distributed software services. In *IFIP/IEEE International Symposium on Integrated Network Management (ISINM'95)*, Santa Babara, USA, 1995.
6. Eclipse Modeling Framework (EMF). http://eclipse.org/emf/.
7. S. Göbel. Encapsulation of Structural Adaptation by Composite Components. In *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04)*, Newport Beach, CA, USA, 2004.
8. S. Göbel and M. Nestler. Composite Component Support for EJB. In *Winter International Symposium on Information and Communication Technologies (WISICT'04)*, Cancun, Mexico, 2004.
9. R. Keller and U. Hölzle. Binary Component Adaptation. In *ECOOP 09*, Brussel, 1998. Springer.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
11. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
12. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Transactions on Software Engineering*, 26(1):70–93, 2000.
13. ObjectWeb. Fractal. http://fractal.objectweb.org/.
14. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.