External Requirements Validation for Component-Based Systems

Andreas Leicher and Felix Bübl

Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
{fbuebl|aleicher}@cs.tu-berlin.de — http://www.cocons.org

Abstract. Software evolution is a major challenge to software development. When adapting a component-based system to new, altered or deleted requirements, existing requirements should not accidentally be violated. Invariant conditions are usually specified via constraint languages like OCL on a high precision level close to source code. On the contrary, this paper uses a new constraint mechanism. One *context-based constraint* (CoCon) specifies one requirement for a group of *indirectly associated* components that share a context. This paper proposes a 'Rule Manager' approach to monitor a system's compliance with requirements automatically at runtime. The approach is compatible with modern middleware technologies and allows the transparent integration of requirement validation in legacy systems or COTS.

1 Introduction

1.1 Continuous Software Engineering

The context for which a software system was designed continuously changes throughout its lifetime. *Continuous software engineering* is a paradigm discussed in [22] to keep track of the ongoing changes and to adapt legacy systems to altered requirements. Only component-based systems are addressed in the KONTENG[1] project, because this rearrangeable software architecture is best suited for continuous software engineering.

1.2 The Notion of 'External Requirements Validation'

Some requirements should be reflected during the design phase, some during deployment and some at runtime. This paper focuses on requirements validation at runtime. We propose to monitor a system's compliance with requirement specifications at runtime. Requirements tend to change quite often. The new approach

[1] This work was supported by the German Federal Ministry of Education and Research as part of the research project KONTENG (Kontinuierliches Engineering für evolutionäre IuK-Infrastrukturen) under grant 01 IS 901 C

A. Banks Pidduck et al. (Eds.): CAiSE 2002, LNCS 2348, pp. 404-419, 2002.

presented in this paper suggests *not* to implement requirements validation *into* any of the components involved. Instead, conformity with requirements is enforced *externally* of the components. Hence, the system can be *transparently* adapted to changed or new requirements via connector refinement without modifying the components directly. Furthermore, legacy components or components 'off the shelf' can be forced to comply with requirements that are not taken into consideration by the components themselves.

2 Context-Based Constraints (CoCons)

This section explains the concept of 'context' used in this paper and introduces how to specify requirements via 'context-based constraints'.

2.1 The New Constraint Technique 'CoCons' in Brief

This article is an overview on the new constraint technique – much more details are provided in the corresponding technical report ([3]). The basic concept can be explained in just a few sentences.

1. Yellow sticky notes are stuck to the components. They are called 'context properties', because meta-information describing their component's context is written on them.
2. A new constraint mechanism refers to this meta-information for identifying the part of the system where the constraint applies. Only those components whose meta-information fits the constraint's 'context condition' must fulfil the constraint. Up to now, no constraint technique exists that selects the constrained components according to their meta-information.
3. Via the new constraint technique a requirement for a group of components that share a context can be automatically protected.

2.2 Introducing Context Properties

The term 'context' is used an various senses. When using context-based constraints, 'context' refers to the state, situation or environment of a component. The context is expressed by assigning context properties to components. A **context property** has a name and a set of values as illustrated in figure 2. One set of values can be assigned to a single component c for each context property cp . This set is called $ConPropVals^{cp,c}$. If a context property value is associated with a component, it describes how or where this component is used – this meta-information describes the 'context' of this component. Each context property is only useful if a requirement must be specified for a *part* of the system that can be identified by this context property. This paper discusses only the context property **'Personal Data'**. It signals whether a component handles data of private nature. Its values associated with a component are defined manually.

On the contrary, **system properties**, like 'the current User' or 'the current IP Address', can be automatically queried from the middleware platform during configuration or at runtime. Each middleware technology enables certain system properties to be queried, but it's beyond the scope of this paper how the various technologies query the system property values.

Many techniques for writing down metainformation exist. The notion of context or container properties is well established in component runtime infrastructures such as COM+, EJB, or .NET. The primary benefit of enriching components with context properties is revealed in next section, where such properties are used to specify requirements.

2.3 A New Notion of Invariants

One requirement can affect several components that do not invoke each other directly or even do not run on the same platform. A **context-based constraint** (CoCon) specifies a requirement for a group of components that share a context. The shared context is identified via the context property values assigned to these components. If these values comply with the CoCon's context condition then their components share the same context. The metamodel in figure 1 shows the metamodel for CoCons. This metamodel resembles a UML 1.4 ([14]) profile if you rename the metaclass 'Component' to 'ModelElement', 'Context Property' to 'TagDefinition' and 'Context Property Value' to 'TaggedValue' in figure 1. However, this paper focuses on using CoCons at runtime. Thus, integration in UML is not discussed. CoCons should be preserved and considered in model modifications, during deployment, at runtime and when specifying another – possibly contradictory – CoCon. Thus, a CoCon is an *invariant*. It describes which parts of the system must be protected. If a requirement is written down via a CoCon, its violation can be detected *automatically*.

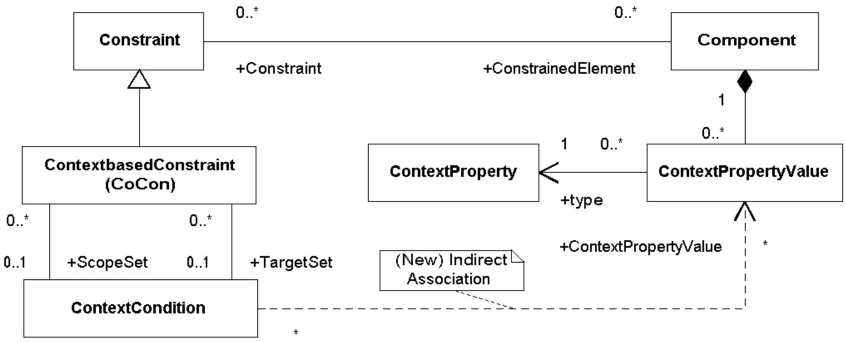

Fig. 1. The CoCon Metamodel

The context property 'Personal Data' is described section 2.2. Either its value 'True' or 'False' is associated with a component. Thus, a CoCon can state that "*components where 'Personal Data' has the value 'True'* must be inaccessible by the component 'SalesMgr"'**(Example A)**. This constraint is based on the context property values of the components – it is a context-based constraint or CoCon. The shared context is expressed via a **'context condition'** which selects components via their context property values. It describes a (possibly empty) *set* of components. In example A, the part in italics represents the context condition that selects the **'target set'**.

In Example A, the *scope* of the CoCon is a single component: the component 'SalesMgr'. But the scope can be a set containing any number of components, as illustrated in **example B**: "*The component 'SalesMgr' must be inaccessible by any component that is currently used by a 'Controller' who is not located in 'Frankfurt'* ". Users of the system have the role 'controller' if they check the financial transactions of accountancy. Both example A and B are explained in section 3.2. The **'scope set'** contains those components, where the elements of the target set must meet the constraint. In Example B, the scope set contains all the components used by a controller not located in Frankfurt. Both target set elements and scope set elements can be specified either directly or indirectly:

...directly: Set elements can be specified by directly naming the component(s) involved. In Example A, the CoCon is associated directly with the 'SalesMgr' component. This unambiguously identifies the only element of the scope set.

...indirectly: The indirect association of a constraint to its constrained elements is the key new concept of context-based constraints. Set elements can be indirectly selected via a context condition. The scope set in Example B contains all the components where the context property 'Current User Role' has the value 'Controller' and 'Current Caller Location' does not equal 'Frankfurt'. These scope set elements are anonymous. They are not directly named, but described indirectly via their context properties. If no component fulfils the context condition, the set is empty. This simply means that the CoCon does not apply to any component.

A CoCon **attribute** can define details of its CoCon. This paper only discusses the attribute **Action**. It defines what to do in case of a CoCon violation. More Details on CoCons are introduced in [4].

2.4 A Textual Language for CoCon Specification

This section introduces a textual language for specifying context-based constraints. The standard technique for defining the syntax of a language is the Backus-Naur Form (BNF), where "::=" stands for the definition, "`Text`" for a nonterminal symbol and "**TEXT**" for a terminal symbol.

The **C***ontext-based* **C***onstraint* **L***anguage CCL for components* ([3]) consists of different CoCon types. However, this paper only discussed one CoCon type. For a complete syntax definition please refer to the implementation of the 'CCL plugin for ArgoUML' described in section 6.1. All rules concerning the separator (',' ; 'AND'or 'OR') are abbreviated. For instance, "`Rule` {**'OR'** `Rule` }*" is abbreviated "`(Rule)`$+^{OR}$". This is the BNF Syntax for CoCons of the InaccessibleBy Type:

`InaccessibleByCoCon`	::=	`(ElementSelection)`$+^{OR}$ **'MUST BE InaccessibleBy'** `(ElementSelection)`$+^{OR}$ `Attribute`
`ElementSelection`	::=	`ContextCondition` | `DirectSelection`
`DirectSelection`	::=	(**'THE COMPONENT'** `CompName`) | **'THIS'**
`ContextCondition`	::=	**'ALL COMPONENTS WHERE'** `ContextQuery`$+^{AND\ or\ OR}$
`ContextQuery`	::=	`ContextPropertyName Condition` (`ContextPropertyValue` | `SetOfConPropValues`)
`SetOfConPropValues`	::=	(**'{'** `(ContextPropertyValue)`$+^{Comma}$**'}'**) | `ContextPropertyName`
`Condition`	::=	**'CONTAINS'** | **'DOES NOT CONTAIN'** | '=' | '! =' | '<' | '>' | '<=' | '>='
`Attribute`	::=	**'WITH ACTION ='** (**'Abort'** | **'Redirect'** | **'Filter'**)

An example is given in section 3.2. The `ContextCondition` rule allows for the *indirect* selection of the elements involved. In contrast, the `ElementName` rule *directly* selects elements by naming them. The `ContextQuery` describes (one or more) set(s) of $RequiredValues^{cp}$. A context condition selects the component c if for each context property cp used in the context condition the $RequiredValues^{cp} \subseteq ConPropVals^{cp,c}$. Besides **'CONTAINS'** ($\subseteq$), this paper suggests other expressions like '**!=**' (does not equal) and **'DOES NOT CONTAIN'** ($\not\subseteq$). Only simple comparisons (inclusion, equality,...) are used in order to keep CoCons comprehensible. Future research might reveal the benefits of using complex logical expression, such as temporal logic.

2.5 Comparing OCL to Context-Based Constraints

Typically, the *Object Constraint Language OCL* summarized in [21] is used for the constraint specification of object-oriented models. One OCL constraint refers to (normally one) *directly identified* element, while a context-based constraint can refer both to directly identified and to (normally many) indirectly identified, *anonymous and unrelated* elements. A CoCon can select the elements involved according to their meta-information. In the UML, tagged values are a mechanism similar to context properties for expressing meta-information. There is no concept of selecting the constrained elements via their tagged values in OCL or any other existing formal constraint language.

An OCL constraint can only refer to elements that are directly linked to its scope. On the contrary, a CoCon scope is not restricted. It can refer to elements that are not necessarily associated with each other or even belong to different models. When specifying an OCL constraint it is not possible to consider elements that are unknown at specification time. In contrast, an element becomes involved in one context-based constraint simply by having the matching context property value(s). Hence, the target elements and the scope elements can change without modifying the CoCon specification.

Before discussing another distinction, the OMG meta-level terminology will be explained briefly. Four levels exist: Level 'M_0' refers to a system's objects at runtime, 'M_1' refers to a system's model or schema, such as a UML model, 'M_2' refers to a metamodel, such as the UML metamodel, and 'M_3' refers to a meta-metamodel, such as the Meta-Object Facility (MOF). If an OCL constraint is associated with a model element on level M_i, then it refers the instances of this model element on level M_{i-1} — in OCL, the 'context' [6] of an invariant is an *instance* of the associated model element. If specified in a system model on M_1 level, an OCL constraint refers to *runtime* instances of the associated model element on level M_0. In order to refer to M_1 level, OCL constraints must be defined at M2 level (e.g. within a stereotype). On the contrary, a CoCon can be verified automatically on the *same* meta-level where it is specified. All CoCons discussed in this paper are specified *and* verified on M_0 level because this paper focuses on checking them at runtime.

3 Applying Context-Based Constraints at Runtime

Today, abstract constraints, like CoCons, are generally not supported by modern component technologies, like Enterprise JavaBeans [7] or Microsoft *.NET*. They have to be implemented manually into components. In fact, finalized components have to be extended and modified by additional logic. They require a programmatic extension that reduces the system's maintainability. This complicates the programming, the handling and the evolution of components. This paper, therefore, proposes to specify constraints declaratively. Most contemporary middleware systems only allow for the specification of constraints for well-defined tasks such as security and transactional behaviour at the time of deployment. Beyond that, the approach presented allows CoCons to be specified and monitored *dynamically* at runtime.

3.1 Main Objectives

The main objective is the transparent integration of a mechanism for monitoring CoCons into modern middleware technologies. This brings the advantage that modifying components is not longer necessary when adapting the system to new or changed requirements expressed via CCL CoCons. In addition, the transparent integration as well as the separation of the requirements specification support the evolution of systems. Requirements are encapsulated as separate aspects of the application logic and are treated separately. Therefore, the approach facilitates the integration with legacy systems, which can be enforced even at runtime to conform to new or changed constraints without modifying the legacy components themselves. Furthermore, components 'of the shelf ' (COTS) can be transparently complemented by additional constraints for the same reasons.

Their standard functionality can be restricted by the approach presented without modifying themselves.

3.2 Scenario

The following example describes a trading system of a company located in New York(main office), Tokyo(branch office) and Frankfurt(branch office). The systems main components are shown in figure 2. They are associated with appropriate context property values. An identical configuration of the system will be deployed and installed at each location. Personal data of local users is stored at the user's location whereas a global user schema exists for the whole system. The following context properties are shown in figure 2:

System.Location: Specifies the installation place of the component

System.UserRole: Describes user groups: In our example, an user is either a Trader or a Controller. Controllers are able to check personal data on all transactions.

In this paper only one requirement is discussed: "*Controllers who are not located in Frankfurt must not be able to access Frankfurt's personal data*". The system must comply with this requirement due to German federal law. The corresponding CoCon specification is defined as follows:

```
ALL COMPONENTS WHERE ‘Personal Data’ = ‘True’
AND System.Location = ‘Frankfurt’
MUST BE InaccessibleBy
ALL COMPONENTS WHERE System.Location != ‘Frankfurt’ AND
System.UserRole = ‘Controller’
WITH ACTION = ‘Abort’
```

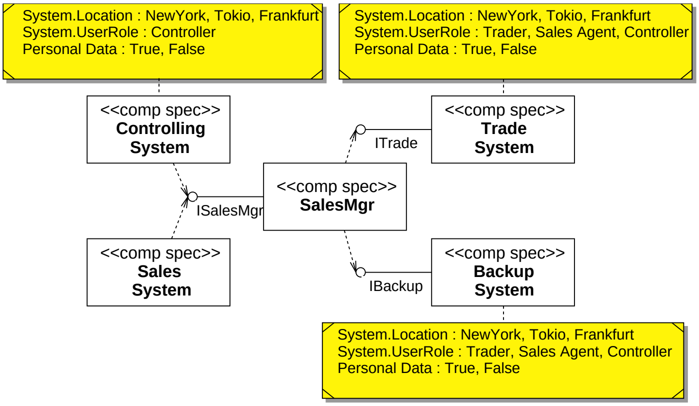
System.Location : NewYork, Tokio, Frankfurt
System.UserRole : Controller
Personal Data : True, False
System.Location : NewYork, Tokio, Frankfurt
System.UserRole : Trader, Sales Agent, Controller
Personal Data : True, False
<<comp spec>>
Controlling System
<<comp spec>>
Trade System
ITrade
<<comp spec>>
SalesMgr
ISalesMgr
<<comp spec>>
Sales System
IBackup
<<comp spec>>
Backup System
System.Location : NewYork, Tokio, Frankfurt
System.UserRole : Trader, Sales Agent, Controller
Personal Data : True, False

Fig. 2. Trader System Scenario

3.3 Runtime Realisation

The example requirement is checked at runtime since it refers to dynamically changing context properties values. The components in question may not be changed because transparent monitoring is strived for. Rather, the communication of the components involved must be intercepted. Hence, monitoring points (Points of Interception) are installed in the communication paths of the components. For this, we use as the technical foundation the proxy pattern[8]. Most middleware technologies use a kind of this standard communication method to realize local and distributed communication.

Figure 3 shows the example scenario once more. This time a J2EE application server is used as the starting point. Requests to the system are routed via the controlling system component to the sales management component. The point of interception is placed, in front of the sales management component. Each incoming call is monitored, for whether it complies with the specified CoCon. In the case a violation against a CoCon, predefined actions are executed.

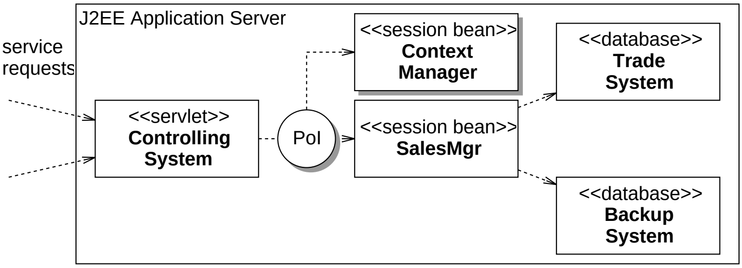
J2EE Application Server
service requests
<<servlet>> Controlling System
PoI
<<session bean>> Context Manager
<<session bean>> SalesMgr
<<database>> Trade System
<<database>> Backup System

Fig. 3. Point of Interception

4 Rule Manager Approach

Our approach is based on two well-known concepts. On the one hand, it uses Event Condition Action (ECA) rules. They are typically applied in active databases [24], where ECA rules are used to check for basic database operations and where additional actions are triggered when specified conditions are met. On the other hand, middleware systems are integrated via a proxy mechanism. It allows for transparent integration in common technologies by intercepting communication between components. The proxy mechanism is extended by the needed monitoring functionality. Thus, it becomes possible to place interception points between components. The extended proxy can be seen as a *connector*, which facilitates requirement protection. Consequently, a connector is treated as a *first-class entity*[19], which has to be carefully designed.

4.1 ECA Rules

The approach presented uses ECA rules to monitor CoCons at runtime. ECA rules are based on a formal specification language and have a syntax depicted in figure 4. A rule consists of four parts: a *type clause*, an *event clause*, a *conditional clause* and an *action clause*[2]. One of the main tasks of this approach is to translate a CoCon into ECA rule definitions.

```
declare rule rule_name
  proxy typeclause
  on eventclause
  if conditionalclause
  do actionclause
end rule
```

Fig. 4. Rule Definition

```
declare rule security_example
  proxy type(SalesMgt)
  on request(true)
  if context(‘PersonalData‘, ‘True‘) AND
¬(context(System.Location, ‘Frankfurt‘)        AND
context(System.UserRole, ‘Controller‘)
  do abort
end rule
```

Fig. 5. Example Rule

CoCons declare invariants between up to two sets of components (see section 2). One CoCon applies to all components whose context property values fulfil its context condition. An ECA rule, on the other hand, references a particular connection between two components. For this reason, a suitable *type clause* must be declared for *each* component affected by the CoCon. A *type clause* specifies the component to which communication has to be filtered. An *event clause* specifies the event that must occur to trigger the ECA rule. An event can be related to a specific, or to several services.

Different types of CoCons exist (though, in this paper only one type relevant to runtime is discussed). These types determine the *conditional clause.* If the constraint is to be evaluated at runtime, as in figure 5, the *conditional clause* contains *context clauses* that determines the matching context property values at runtime. A *context clause* is related to the Context Manager, which provides the context property values of a component. The context manager detects this information through technology-specific functions. Every message (or procedure call) to a certain proxy is checked by an ECA rule at runtime. In case of a conditional match, there are many possible actions. These can be stated in the *actions clause.* Actions are managed in an *action template* repository that contains some predicates with platform specific mappings.

4.2 Actions

The specification of a runtime CoCon includes the action that must be taken in the case where a CoCon is violated. There are many possible actions. A selection of typical actions follows without claiming to be complete:

[2] In the following, clauses are stated as predicates

Abort Action The communication is prevented specifically. This can be realized by sending back an exception to the calling client. This is totally transparent to the application components.

Redirect Action: The communication, which is normally initiated between components $A \rightarrow B$, will be redirected to an available and fully compatible component C ($A \rightarrow C$). Component C has to fulfil the contract of component B with A.

Filter Action: The data transferred in communication calls have to be modified. Therefore, the exact knowledge of the format used for the submitting information is necessary.

Context-Aware Service: Besides normal communication, additional actions will be executed. For example, log-entries or some kind of analysis could be transparently integrated into communication.

4.3 Calculating the Points of Interception

The monitoring of CoCons at runtime requires that monitoring points (interception points) be determined. As a starting point of the calculation, there are both a component specification model as known from Cheesman/Daniels[5] and an abstract requirements (CoCon) specification. The component specification model has been enriched by context properties. The following problems arise in the context of the calculation:

- CoCons can specify requirements for sets of components. Thus, a large number of ECA rules can result from a single CoCon. Because of the nature of CoCons components can be directly or transitively associated. The resulting ECA rules must be installed at each possible connection path between the sets of components. This ensures that the communication between components can be suitable monitored.
- Interception points between components that are not connected directly can be installed in different places. The problem therefore arises to localize the best suitable point.

In the following possible solutions to these problems are discussed briefly.

Invocation Path Calculation The component specification model (which shows dependencies between components) can be used to calculate all invocation paths of the system. A sequence of components that are directly connected with each other is referred as an invocation path. Figure 6 shows a part of the component specification diagram from the scenario, including context properties. Additionally, a CoCon is indicated. CoCons are build up out of two sets, the target set and the scope set. All components, which belong to one of these sets, are involved in the current CoCon. In order to determine the interception points, all of the invocation paths between these components must be identified. Therefore, direct and transitive dependencies between all the components in each set have to be calculated. As a result. all invocation paths have to fulfil the following predicates: They start with an element of the scope set, they end with an element of the target set, and all elements between them are connected directly via invocation dependencies.

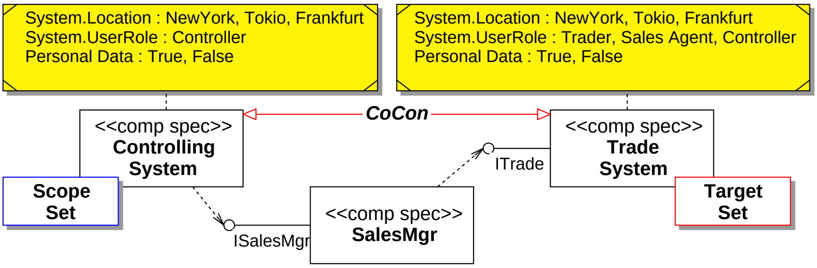
System.Location : NewYork, Tokio, Frankfurt
System.UserRole : Controller
Personal Data : True, False
System.Location : NewYork, Tokio, Frankfurt
System.UserRole : Trader, Sales Agent, Controller
Personal Data : True, False
CoCon
<<comp spec>> Controlling System
<<comp spec>> Trade System
<<comp spec>> SalesMgr
Scope Set
Target Set
ITrade
ISalesMgr

Fig. 6. Invocation path Calculation

Identifying the Interception Point An interception point has to be determined for each invocation path. This is a complicated task because the integrity and the performance of the system have to be maintained. Currently, we follow the heuristic to place the interception point in front of the serving component (service provider). This has the advantage of guaranteeing that all communication calls can be intercepted. In this way, we maintain the integrity of the system but haven't optimised the intercepting mechanism.

4.4 System Structure

As explained above, CoCons must be translated into ECA rules that have to be assigned to appropriate proxies. These tasks are reflected in the structure of the Rule Engine (see figure 7), which consists primarily of two parts:

The Rule Compiler translates CCL constraints into formal ECA rule specifications for each proxy involved. Context information as well as *context-templates* assist the compiler in translating the CCL constraint. Context-templates enable the necessary information to be determined at runtime. In order to install the ECA rules at proxies, they are compiled to executable Java code called *plug-ins* and dynamically uploaded to corresponding proxies. Proxies are enabled to update and execute these plug-ins.

The Rule Manager is the central management component of the engine. It manages the ECA rules in a repository and is responsible for updating the proxies in case of changes. The rule manager registers all proxies in the repository and transmits applicable ECA rules only to the components concerned.

The main problem of this approach arises from the determination of the context property values of the components involved in the checked communication call.

Current values can be extracted from the call protocol in question, from the environment (e.g. the application server) or even from the client. The context manager(see figure 3) serves as a central repository that provides the current context property values of each component.

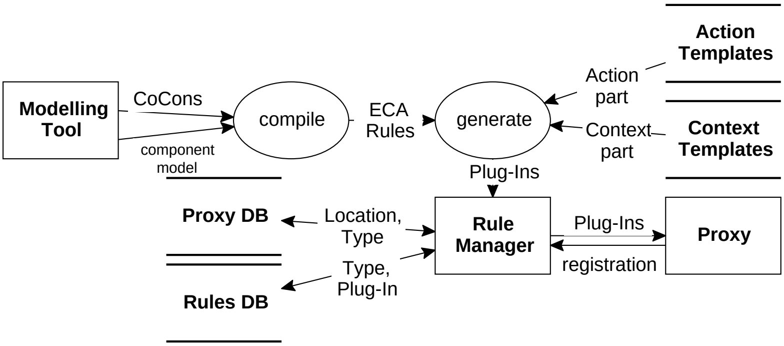

Fig. 7. System Structure Overview

5 Integration Approach

The integration of a CoCon monitoring mechanism into middleware systems can be realized through different concepts. The main difficulty is the incorporation of the extended proxy mechanism into existing wiring standards of middleware technologies.

5.1 System-Level Integration

The underlying middleware technology is transparently modified via the extension mechanism proposed. Therefore, the middleware's source code, especially of the proxy mechanism, has to be modified. At a minimum, the interception call has to be inserted into the code, but also the whole extension mechanism can be integrated, as well. Figure 8 shows the intended basic concept. The RPC/RMI functionality of the J2EE Application Server is thereby enlarged. Obviously, components at the application level communicate directly with each other. At the system level, however, communication is controlled by a proxy mechanism, which realizes remote connection and services like transactionality and security. Proxies are currently examined in various scientific works such as [2,18,17]. To customize transparent integration, some middleware platforms offer *callback* functions to modify standard behaviour. The JBoss Server ([15]), for example, supports a callback interface to intercept method calls to Enterprise JavaBeans. Such extensions simplify setting up integration approaches. The BEA WebLogic Server [1], for example, supports a callback interface in order to specify customized load balancing. In addition, Microsoft offers several integration techniques that easily and transparently allow adding intercepting functions to existing software. For example, so-called *Hooks* can be used to monitor relevant events and to call declared functions to handle these events ([9]). At the moment, a prototype based on JBoss is being developed. It cannot fulfil all advantages due to limited interception interfaces. However, we hope to achieve conclusions regarding the performance and the effectiveness of system-level integration.

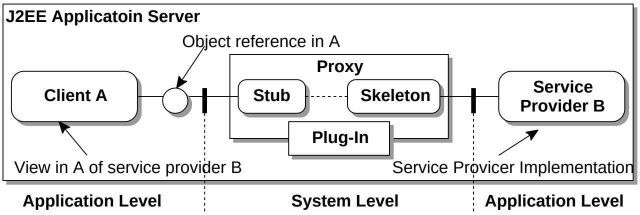

Fig. 8. Proxy Mechanism

5.2 Application-Level Integration

At the application level, there are several possibilities for obtaining the needed monitoring functionality. Most techniques are based on a wrapping concept for integrating additional functionality to existing concepts. Some wrapping concepts are represented in illustration 9. The standard approach is shown as variant 1. It is based on a proxy component, which is put in front of the service provider component. The advantage of this approach is that application level components do not have to be modified. It is a transparent integration approach. In contrast to the same principle on the system level, however the control is limited. For example, the Self problem (disclosure of the component's identity)[23] cannot be fully prevented and even worse, not all contexts can be evaluated. The second variant shows the inheritance of the application level component. This means that the source code of the component is needed and has to be extended to contain monitoring functionality. This approach has not been investigated further at this time. The third variant shows an aggregation approach. It is similar to the first approach, but merges the proxy and application component. Therefore, changing the source code is normally needed here, as well. In contrast to the second version, however, the component's classes could be repacked in Java without directly modifying the code. At present, we also use this concept in our research.

An alternative technology comparable to wrapping is not shown in the figure: Aspect-oriented programming[10]. Aspects cleanly encapsulate crosscutting concerns. Consequently, the extended proxy mechanism could be declared as an aspect and precompiled into the components involved.

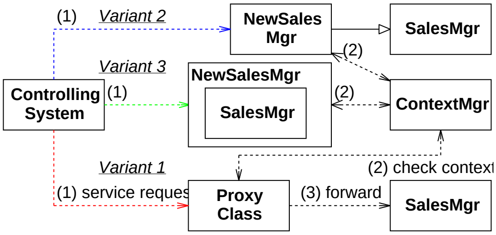

Fig. 9. Integration Variants

6 Conclusion

6.1 Applying CCL in Different Development Levels

This paper focuses on checking CCL CoCons at runtime. However, in this section the application of CCL throughout the software development process is outlined.

During requirements analysis the business experts must be asked specific questions in order to find out useful context properties and CoCons. Currently a CCL-aware method for requirements analysis is being developed at the Technical University (TU) of Berlin in cooperation with Eurocontrol, Paris.

The benefits of considering both requirements and architecture throughout the **design phase** are discussed in [13]. The application of CLL during design is currently being evaluated in a case study being carried out in cooperation with the ISST Fraunhofer Institute, the TU Berlin and the insurance company Schwäbisch Hall. In winter term 2001/02 a 'CCL plugin' for the open source UML editor 'ArgoUML' has been implemented at the TU Berlin to automatically protect CCL specifications during design. It is available for download at `ccl-plugin.berlios.de`.

The people who want to enforce a new requirement often don't know the details of every part of the system, and neither do they have access to the complete source code. By using CoCons, developers don't have to understand every detail ('glass box view') or modify autonomous parts of the system in order to enforce a new requirement on them. Instead, context properties can be assigned *externally* to an autonomous component, and the communication with this component can be monitored *externally* for whether it complies with the CoCon specification **at runtime**. A prototypical rule manager framework as described in this paper currently is integrated into an application server at application level in cooperation of the TU of Berlin with BEA Systems and the Fraunhofer ISST. Thus, legacy components or components 'off the shelf' can be forced to comply with new requirements.

6.2 Limitations of Context-Based Constraints

Regarding only the context properties of a component bears some risks. It is crucial that theses values are always up-to-date. System properties, however, always have the current value, because they are queried from the middleware platform at runtime. Only one ontology should be used within a single system. For example, the context property 'Personal Data' should have exactly this name in every part of the system, even if these parts are manufactured by different companies. A common ontology can be ensured by either demanding it from the manufacturers or by intellectually inspecting an 'off the shelf' or legacy component when integrating it into the system.

6.3 Benefits of Context-Based Constraints

Similar to context properties, [16] suggest assigning metadata to components in order to use it for different tasks throughout the software engineering lifecycle. Likewise, [20] annotate components in order to perform dependency analysis over these descriptions. Other concepts for considering metadata exist, but none of them writes down *constraints that reflect this metadata* as introduced here. Key requirements can now be expressed according to the system's context. They can be specified in an easily comprehensible, straightforward language.

Maintenance is a key issue in *continuous software engineering* ([12]). CoCons facilitate consistency in system evolution. They assist in detecting when software or context modifications compromise intended functionality. Requirements tend to change quite often. The Rule Manager approach enables a system to be adapted to changed to new requirements via connector refinement without modifying the components. Thus, legacy components or 'off the shelf' components can be forced to comply with requirements that are not taken into consideration by the components themselves. The adaptability of a system is improved by enforcing conformity with meta-information. This meta-information can be easily adapted, whenever the context of a component changes. Furthermore, CoCon specifications themselves can be modified, as well, if the requirements change. Each deleted, modified or added CoCon can be automatically enforced, and resulting conflicts can be identified automatically as described in [3]. It is changing contexts that drive evolution. CoCons are context-based and are therefore easily adapted if the contexts, the requirements or the configuration changes. The compliance of a system with requirements can now verified automatically. According to [11], automated support for software evolution is central to solving some very important technical problems in current day software engineering.

References

1. BEA Systems, Inc. *BEA WebLogic Server, Using WebLogic Server Clusters*, March 2001.
2. 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*, Reading, 2000. Addison-Wesley.
3. Felix Bübl. The context-based constraint language CCL for component. Technical report, Technical University Berlin, available at www.CoCons.org, 2002.

4. Felix Bübl. Introducing context-based constraints. In Herbert Weber and Ralf-Detlef Kutsche, editors, *Fundamental Approaches to Software Engineering (FASE ' 02), Grenoble, France.* Springer, April 2002.
5. John Cheesman and John Daniels. *UML Components.* Addison-Wesley, 2000.
6. Steve Cook, Anneke Kleppe, Richard Mitchell, Jos Warmer, and Alan Wills. Defining the context of OCL expressions. In B.Rumpe and R.B.France, editors, *2nd International Conference on the Unified Modeling Language, Colorado, USA*, volume 1723 of *LNCS.* Springer, 1999.
7. Linda G. DeMichiel, L. Umit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification.* Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A., April 2001. Proposed Final Draft.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. Yariv Kaplan. API spying techniques for windows 9x, NT and 2000. http://www.internals.com/articles/apispy/apispy.htm, 2001.
10. 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.
11. Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.
12. Hausi Müller and Herber Weber, editors. *Continuous Engineering of Industrial Scale Software Systems*, Dagstuhl Seminar #98092, Report No. 203, IBFI, Schloss Dagstuhl, March 2-6 1998.
13. Bashar Nuseibeh. Weaving the software development process between requirements and architecture. In *Proceedings of ICSE-2001 International Workshop: From Software Requirements to Architectures (STRAW-01) Toronto, Canada*, 2001.
14. OMG. UML specification v1.4 (ad/01-02-14), 2001.
15. JBoss Organization. Jboss website. http://www.jboss.org, December 2001.
16. Alessandro Orso, Mary Jean Harrold, and David Rosenblum. Component metadata for software engineering tasks. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, volume 1999 of *LNCS*, Berlin, November 2000. Springer.
17. G. S. Reddy and R. K. Joshi. Filter objects for distributed object systems. *Journal of Object Oriented Programming*, 13(9):12–17, January 2001.
18. 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.
19. Mary Shaw and David Garlan. *Software Architecture.* Prentice-Hall, 1996.
20. Judith A. Stafford and Alexander L. Wolf. Annotating components to support component-based static analyses of software systems. In *Grace Hopper Celebration of Women in Computing, Hyannis, Massachusetts*, September 2000.
21. Jos B. Warmer and Anneke G. Kleppe. *Object Constraint Language – Precise modeling with UML.* Addison-Wesley, Reading, 1999.
22. Herbert Weber. Continuous engineering of information and communication infrastructures (extended abstract). In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering FASE'99 Amsterdam Proceedings*, volume 1577 of *LNCS*, pages 22–29, Berlin, March 22-28 1999. Springer.
23. Ian Welch and Robert J. Stroud. From Dalang to Kava - the evolution of a reflective java extension. In *Reflection*, pages 2–21, 1999.
24. Jennifer Widom and Umeshwar Dayal. *A Guide To Active Databases.* Morgan-Kaufmann, 1993.