Introducing Context-Based Constraints

Felix Bübl

Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
fbuebl@cs.tu-berlin.de
http://www.CoCons.org

Abstract Software evolution is a major challenge to software development. When adapting a system model to new, altered or deleted requirements, existing requirements should not unintentionally be violated. One requirement can affect several possibly unassociated elements of a system. A new constraint technique is introduced in this paper: One *context-based constraint* (CoCon) specifies a requirement for those system (model) elements that belong to the related context. The constrained elements are indirectly selected via their meta-information. Thus, verifying compliance with requirements can be supported automatically when a system's model is modified, during (re-)configuration and at runtime.

1 Introduction: Continuous Engineering

1.1 Continuous Engineering Requires 'Design for Change'

The context for which a software system was designed changes continuously throughout its lifetime. **Continuous software engineering** (CSE) is a paradigm discussed in [18] for keeping track of the ongoing changes and to adapt legacy systems to altered requirements as addressed in the KONTENG[1] project. The system must be prepared for adding, removing or changing requirements. The examples in this paper concentrate on component-based software systems because this rearrangeable software architecture is best suited for CSE.

New methods and techniques are required to ensure consistent modification steps in order to safely transform the system from one state of evolution to the next without unintentional violating existing dependencies or invariants. This paper focuses on recording requirements via constraints in order to protect them from unwanted modifications. However, an enhanced notion of 'constraint', introduced in section 3, is needed for this approach.

1.2 Focus: Requirements Specification in System Models

This paper proposes to express important requirements via a new specification technique that facilitates their consideration in different levels of the software

[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.

R.-D. Kutsche and H. Weber (Eds.): FASE 2002, LNCS 2306, pp. 249–263, 2002.

development process: some requirements should be reflected in models, some during coding, some during deployment and some at runtime. This paper focuses on specifying requirements in models. Thus, this paper discusses how to write down which *model elements* are affected by a requirement. Obviously, there are no *model* elements during configuration or at runtime. When applying the new approach discussed here during configuration or at runtime, please read model element as *system element during configuration* or *system element at runtime* throughout the paper.

1.3 The New Concept in Brief

The basic idea introduced in this paper can be explained in just a few sentences.

1. Yellow sticky notes are stuck onto the model elements. They are called 'context properties'because they describe the context of their model element.
2. A new constraint mechanism refers to this meta-information for identifying the part of the system where the constraint applies. Only those model elements whose meta-information fits the constraint's 'context condition' must fulfill the constraint. Up to now, no constraint technique exists that selects the constrained elements according to their meta-information.
3. Via the new constraint technique a requirement for a group of model elements that share a context can be protected automatically in system modifications.

This article is an overview on the new constraint technique – much more details are provided in the corresponding technical report ([2]).

2 Introducing Context Properties

This section explains the concept of 'context' used here.

2.1 Describing Indirect Dependencies via Context Properties

A **context property** has a name and a set of values. A formal definition is given in [2]. If its values are assigned to an element, they describe how or where this element is used – they show the context of this element. The name of the context property stays the same when assigning its values to several elements, while its values might vary for each element. For example, the values of the context property 'Workflow' reflect in which workflows the associated element is used, as discussed in section 2.2. A graphical representation is indicated in figure 1. The context property symbol resembles the UML symbol for comments because both describe the model element they are attached to. The context property symbol is assigned to one model element and contains the name and values of one context property specified for this model element. However, it is also possible to use one context property symbol for each context property that is assigned to the same model element. The primary benefit of enriching model elements with context properties is revealed in section 3, where such properties are used to specify requirements.

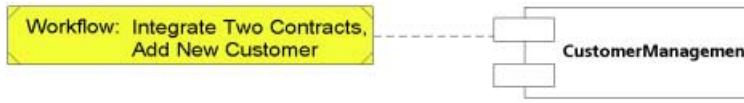

Figure 1. The Context Property Symbol

2.2 General Context Properties

Only three context properties are presented here. The proposed context properties may be ignored and others might be used by the developer as needed in the application domain.

'Workflow' reflects the most frequent workflows and enables the designer to write down requirement specifications for them. If preferred, the term 'Business Process' or 'Use Case' may be used instead of 'Workflow'. For example, a requirement in a system could state that *"all classes needed by the workflow 'Integrate Two Contracts' must be unreadable by the 'Web Server' component"*. This requirement can be written down by identifying all of the classes involved accordingly. This paper suggests taking only the names of the workflows used most often into account for requirement specification. Hiding avoidable granularity by only considering *static aspects of behavior* (= nothing but workflow names) enables developers to ignore details. Otherwise, the complexity would get out of hand. The goal of this paper is to keep the requirement specifications as straightforward as possible.

'Personal Data' signals whether a model element handles data of private nature. Thus, privacy policies, like, *"all components handling Personal Data must be unreadable by the components belonging to the workflow 'Calculate Financial Report' "* can be specified.

'Operational Area' allows for the specification of requirements for certain departments or domains in general, like *"all components handling personal data must be unreadable by all components belonging to the operational area 'Controlling"'*. It provides an organizational perspective.

2.3 Belongs-To Relations

Elements can belong to each other. If, for instance, the model element e is a package, then all the model elements inside this package belong to e. In this case, a context property value assigned to e is automatically assigned to all elements belonging to e. A Belongs-To relation is a directed, transitive relation between elements. One set of values can be assigned to a single element e for each context property *cp* . This set is called $ConPropVals^{cp,e}$. The Belongs-To relation of the element e_{owner} of the 'Owner'-type to other elements $e_{i,\ldots,j}$ of the 'Part'-type is represented via '**Part** $\overset{BeTo}{\longrightarrow}$ **Owner**'. A Belongs-To relation has

the following impact: $\forall i \leq n \leq j : ConPropVals^{cp,e_{owner}} \subseteq ConPropVals^{cp,e_n}$ — all values of *cp* assigned to e_{owner} are also assigned to $e_{i,\ldots,j}$.

The values $ConPropVals^{cp,e_{owner}}$ are *'associated with'* e_{owner}, and *'assigned to'* e_{owner} and all $e_{i,\ldots,j}$ due to the Belongs-To relation. The term 'associated with' is only used for the root element e_{owner}. When implementing a context-property-aware tool, like a modeling tool, only the *associated* values must be made persistent because the derived values can be derived from the associated values as needed.

Some Belongs-To relations are *implicit*. An implicit Belongs-To Relation $e_1 \overset{BeTo}{\longrightarrow} e_2$ exists between the elements e_1 and e_2, if e_1 is part of e_2. For example, all model elements *inside* a package implicitly belong to this package. No model element inside this package does not belong to this package. The fact that e_1 is part of e_2 usually is specified either as composition or aggregation in UML. According to the modeling approach used, other implicit Belongs-To relations can exist. On the contrary, some Belongs-To relations are *explicit*. They must be manually defined, as discussed in [2].

Belongs-To relations create a hierarchy of context property values because they are transitive: if $a \overset{BeTo}{\longrightarrow} b \overset{BeTo}{\longrightarrow} c$ then $a \overset{BeTo}{\longrightarrow} c$. Thus, a context property value associated with c automatically is assigned to b and a. This Belongs-To hierarchy provides a useful structure. It enables the designer to associate a context property value with the element that is as high as possible in the Belongs-To hierarchy. It must be associated only once and usually applies to many elements. Hence, redundant and possibly inconsistent context property values can be avoided, and the comprehensibility is increased.

2.4 Additional Features of Context Properties

This section briefly outlines two more features of context properties. Details are explained in [2]. The context property value assigned to an element can also depend on other influences. For instance, the value can depend on the current state at runtime or on other context property values assigned to the same element.

Usually, the context property values assigned to an element have to be defined manually. Nevertheless, values of **system properties**, like 'the current user' or 'the current IP-address', can only be queried from the middleware platform during configuration or at runtime. This paper focuses on *semantic* context properties that are not automatically available due to the underlying middleware platform.

2.5 Research Related to Context Properties

Many techniques for writing down meta-information exist The notion of context or container properties is well established in component runtime infrastructures such as COM+ EJB, or .NET. Context properties are similar to tagged values in UML - on the design level, tagged values can be used to express context properties. In contrast to tagged values, the values of a context property must

fulfill some semantic constraints. For example, the values of one context property for one model element are not allowed to contradict. E.g., the values of 'Personal Data' must not be both 'Yes' and 'No' for the same model element. Furthermore, not every value may be allowed for a context property. For instance, only the few names of existing workflows are valid values of the context property Workflow.

UML diagrams can express *direct* dependencies between model elements via associations. In contrast, context properties allow the specification of *indirect* dependencies between otherwise unassociated model elements that share the same context. A context property groups model elements that share a context. Existing grouping mechanisms like *inheritance*, *stereotypes* ([1]) or *packages* are not used because the values of a context property associated with one model element might vary in different configurations or even change at runtime. The instances of a model element are not supposed to change their stereotype or package during (re-)configuration or at runtime. One context property can be assigned to different types of model elements. For example, the values of 'Workflow' can be associated both with 'classes' in a class diagram and with 'components' in a component diagram. Using packages or inheritance is not as flexible. According to [13], stereotypes can group model elements of different types via the baseClass attribute, too. However, this 'feature' has to be used carefully and the instances of an model element are not allowed to change their stereotype. Context properties are a simple mechanism for grouping otherwise possibly unassociated model elements - even across different views or diagram types.

3 Introducing Context-Based Constraints (CoCons)

This section presents a new constraint technique for requirements specification.

3.1 A New Notion of Invariants

One requirement can affect several possibly unassociated model elements in different diagrams. A **context-based constraint** (CoCon) specifies a requirement for a group of model elements that share a context. The shared context is identified via the context property values assigned to these elements. If these values comply with the CoCon's context condition then their elements share same context. The metamodel in figure 2 shows the abstract syntax for CoCons. The metaclasses 'ModelElement' and 'Constraint' of the UML 1.4 'core' package used in figure 2 are explained in [13].

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* as described in section 3.5.

As proposed in section 2.2, a context property 'Workflow' is assigned to each model element. Thus, a CoCon can state that "*All classes belonging to the workflow 'Integrate Two Contracts' must be unreadable by the component*

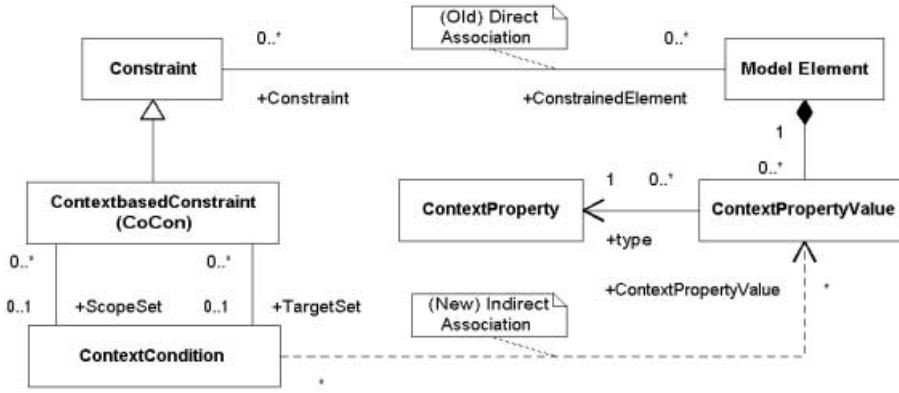

Figure 2. The CoCon Metamodel

'Customer Management'" **(Example A)**. This constraint is based on the context of the classes – it is a *context-based constraint*. Another requirement might state that "*the class 'Employee' must be unreadable by any class whose context property 'Operational Area' contains the value 'Field Service'* " **(Example B)**.

3.2 New: Indirect Selection of Constrained Elements

One CoCon applies to model elements that share a context. The shared context is expressed using a **'context condition'** that selects model elements via their context property values. It describes a (possibly empty) *set* of model elements. A context condition may be restricted to model elements of one metaclass – in examples A and B, the context condition is restricted to 'classes' – no model elements of other metatypes, e.g. 'components', are selected even if their context property values otherwise fit the context condition. A **range** can limit the number of model elements that are denoted by a context condition. The range mechanism is not discussed in this paper – it is needed to specify 'flexible CoCons'. Two different kinds of sets can be selected by a context condition:

On the one hand, a context condition can determine the **'target set'** containing the elements that are checked by the CoCon. In example A, the target set is selected via "*all classes belonging to the workflow 'Integrate Two Contracts'*". On the other hand, a context condition can select the 'scope set' that represents the *part of the system*, where the CoCon is enforced. In example A, the scope of the CoCon is a single model element – the component 'Customer Management'. Nevertheless, the scope of a CoCon can be a **'scope set'** containing any number of elements, as illustrated in example B.

Both target set elements and scope set elements can be selected either directly or indirectly: Set elements can be *directly* associated to a CoCon by naming the model element(s) or by using the keyword `this` as in OCL. In example A,

the CoCon is associated *directly* with the 'Customer Management' component. This unambiguously identifies the only element of the scope set. The new key concept of context-based constraints is **indirect association**. Set elements can be *indirectly associated* with a CoCon via a context condition. The scope set in example B contains all the classes whose context property 'Operational Area' contains the value 'Field Service'. These scope set elements are anonymous. They are not directly named or associated, but described indirectly via their context property values. If no element fulfills the context condition, the set is empty. This simply means that the CoCon does not apply to any element at all. This 'indirect association' is represented as a dotted line in fig. 2 because it is not a UML association with an AssociationStart and an AssociationEnd. Instead, it is a UML dependency. The 'indirectly associated' model elements are selected by evaluating the context condition each time when the system is checked for whether it complies with the CoCon.

As explained in section 2.3, the Belongs-To relation is transitive: if $a \overset{BeTo}{\longrightarrow} b \overset{BeTo}{\longrightarrow} c$ then $a \overset{BeTo}{\longrightarrow} c$. If the context property value v is *associated* with the element e then **transitive closure** $BeTo^*_{v,e}$ contains all elements where v is assigned to due to a Belongs-To relation. If v is associated with c in $a \overset{BeTo}{\longrightarrow} b \overset{BeTo}{\longrightarrow} c$ then $BeTo^*_{v,c} = \{a, b\}$ (if no other elements than a and b belong to c). A context condition selects an element if the context property values *assigned to* this element ($ConPropVals^{cp,e}$) match the context condition. The transitive closure $BeTo^*_{v,e}$ must be considered when evaluating a context condition. Many algorithms exist for calculating a transitive closure. A well-known one is the Floyd-Warshall algorithm. It was published by Floyd ([8]), and is based on one of Warshall's theorems ([17]). Its running time is cubed in the number of elements.

A CoCon can have two context conditions in different roles: one describing *which* model elements are controlled by the CoCon (contained in the target set), and one describing *where (In which parts of the system? Only in the scope set)* the target elements are checked for whether or not they comply with the CoCon. Yet, in some cases the names 'target set' and 'scope set' do not seem appropriate. Mixing example A and example B, a CoCon could state that "*All classes belonging to the workflow 'Integrate Two Contracts'* (Set_1) must be unreadable by *all classes whose context property 'Operational Area' contains the value 'Field Service'* (Set_2)". In this paper, Set_2 is called the scope set. Nevertheless, which part of the system is the scope in this example? Should those elements of the system be called 'scope' which are unreadable(Set_1), or does 'scope' refer to all elements (in Set_2) that cannot read the elements in Set_1? Should Set_1 be called 'scope set one' and Set_2 'scope set two' or is it better to name one of them 'target set'?Unfortunately, there is no intuitive answer yet. Perhaps better names will be invented in the future. But, for most CoCon types (see [2]) the names 'target set' for Set_1 and 'scope set' for Set_2 fit well.

3.3 Handling Conflicts within CoCon Type Families

In this paper, only CoCons of the '**UnreadableBy**' type are discussed due to space limitations. They specify that the target set elements cannot be accessed by the CoCon's scope set elements. Many other CoCon types are discussed in [2]. A *CoCon type family* groups CoCon types. For instance, the UnreadableBy CoCon belongs to the family of Accessibility CoCon types. Conflicting requirements can be automatically detected via *CoCon family specific constraints*. In the case of UnreadableBy CoCons, the CoCon family specific constraints are:

1. No element of the target set may be both ReadableBy and UnreadableBy any element in the scope set.
2. No model element of the target set may be UnreadableBy itself.

One kind of context conditions exists that doesn't refer to context property values: the **joker** context condition simply selects *all* model elements regardless of their context. A context-based constraint is called **simple**, if either its target set or its scope set contains *all* elements of the whole system via an unrestricted joker condition or if it contains exactly one directly associated element. Conflicts may arise if several CoCons of the same CoCon type family apply to the same element. Defining a CoCon's **priority** can prevent these conflicts. If several CoCons of the same CoCon type family apply to the same model element then only the CoCon with the highest priority is checked. If this CoCon is invalid because its scope set is empty then the next CoCon with the second-highest priority is checked. The value of the priority attribute should be a number. This paper does not attempt to discuss priority rules in detail, but it offers the following suggestion: a **default CoCon** which applies to all elements where no other CoCon applies should have the lowest priority. Default CoCons can be specified using a joker condition as introduced in section 3.1. CoCons using two context conditions should have *middle priority*. These constraints express the basic design decisions for *two possibly large* sets of elements. CoCons with one context condition have *high priority* because they express design decisions for *one possibly large* set of elements. CoCons that select both the target set and the scope set *directly* should have the *highest priority* – they describe exceptions for some individual elements.

A CoCon **attribute** can define details of its CoCon. Each attribute has a name and one or more value(s). This paper only discusses two general attributes that can be applied to *all* CoCon types: A CoCon can be named via the attribute **CoConName**. This name must be unique because it is used to refer to this CoCon. Moreover, the **Priority** of a CoCon can be defined via an attribute.

3.4 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.

This paper only discusses one CoCon type: the UnreadableBy CoCons enforce access permission. The model elements in its target set are unreadable by all the model elements in its scope set. In the BNF rules, 'UnreadableBy' is abbreviated '`UR`'. Furthermore, all rules concerning concatenation via a separator (',' 'OR' or 'AND') are abbreviated: "`(Rule)`$+^{Separator}$" represents "`Rule {Separator Rule }*`".

`URCoCon`	::=	`URElementSelection`$+^{OR}$ **'MUST BE UnreadableBy'** `URElementSelection`$+^{OR}$ [**'WITH'** `URAttribute`$+^{AND}$`]`
`URElementSelection`	::=	`URContextCondition` | `URDirectSelection` | **'THIS'**
`URDirectSelection`	::=	**'THE'** `URRestriction ElementName`
`URContextCondition`	::=	`Range (URRestrictions` | **'ELEMENTS'**`)` [**'WHERE'** `ContextQuery`$+^{AND\ or\ OR}$`]`
`Range`	::=	**'ALL'** | `Number` | **'['** `LowerBoundNumber` **','** `UpperBoundNumber` **']'**
`ContextQuery`	::=	`ContextPropertyName Condition (ContextPropertyValue` | `SetOfConPropValues)`
`SetOfConPropValues`	::=	(**'{'** `(ContextPropertyValue)`$+^{Comma}$**'}'**) | `ContextPropertyName`
`Condition`	::=	**'CONTAINS'** | **'DOES NOT CONTAIN'** | **'='** | **'!='** | **'<'** | **'>'** | **'<='** | **'>='**
`URAttribute`	::=	(**'CoConNAME =**' `Name`) | (**'PRIORITY =**' `PriorityValue`)

The `URContextCondition` rule allows for the *indirect* selection of the elements involved. In contrast, the `ElementName` rule *directly* selects elements by naming them. The `URRestriction(s)` rules depend the CoCon type and on the modeling approach used – [2] defines them as "**'Components'** | **'Interfaces'**`)+`" for the 'UML Components' approach ([4]).

The `ConditionExpression` describes (one or more) set(s) of $RequiredValues^{cp}$. A context condition selects e, if for each context property cp used in the context condition the $RequiredValues^{cp} \subseteq ConPropVals^{cp,e}$. 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.

Different kinds of context conditions can be defined in *simple CoCons*. On the one hand, a simple CoCon can have a joker condition instead of a context condition. A joker condition can be defined by omitting the '`WHERE ...`' clause in the `URContextCondition` rule. For instance, '*ALL COMPONENTS* `MUST BE ...`' is a joker condition. On the other hand, a simple CoCon can be specified via the terminal symbol '`THIS`' in the `URElementSelection` rule. '`THIS`' in CCL has the same semantic as '`this`' in OCL (see [5, 16]). If the CoCon is directly

associated with an model element via an UML association then 'THIS' refers to this model element.

3.5 'Privacy Policy' Example of Using UnreadableBy CoCons

In this section, an UnreadableBy CoCon illustrates the case where the target set and the scope set of a CoCon can overlap. Moreover, the example shows how to detect whether a system model complies with a CoCon. A requirement might state that *"All components belonging to the operational area 'Controlling' are not allowed to access components that handle personal data"*. This statement can be specified as a CoCon in this way:

```
ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes'
MUST BE UnreadableBy
ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'
```

If a component has the value 'Yes' in its context property 'Personal Data' *and* the value 'Controlling' in its context property 'Operational Area' then it belongs *both* to the target set and to the scope set of the CoCon. This is absurd, of course. It means that this component cannot read itself. The CoCon is valid, but the system model does not comply with this CCL specification. Such bad design is detected via the CoCon type family specific constraint number two in section 3.3. Every component involved in this conflict must be changed until *either* handles personal data *or* belongs to the 'Controlling'. It must not belong to *both* contexts. If it is not possible adjust the system accordingly, then it cannot comply with the requirement.

3.6 Present Research Results

A CoCon language consists of different CoCon types. Up to now, two CoCon languages exist: The **D**istribution **C**onstraint **L**anguage **DCL** supports the design of distributed systems as described in [3]. It was developed in cooperation with the Senate of Berlin, debis and the Technical University of Berlin. DCL concepts have been implemented at the Technical University of Berlin by extending the tool 'Rational Rose'. A prototype is available for download in German only. It turned out that Rose's extensibility is inadequate for integrating DCL concepts.

The context-based **C**omponent **C**onstraint **L**anguage **CCL** introduced in [2] consists of CoCon types that describe requirements within the logical architecture of a component-based system. It is currently being evaluated in a case study undertaken in cooperation with the ISST Fraunhofer Institute, the Technical University of Berlin and the insurance company Schwäbisch Hall. The UnreadableBy CoCon discussed in this paper is one of CCL's many CoCon types.

UML *profiles* provide a standard way to use UML in a particular area without having to extend or modify the UML metamodel. A profile tailors UML for a specific domain or process. It does not extend the UML by adding any new basic concepts. Instead, it provides conventions for applying and specializing standard UML to a particular environment or domain. Hence, a UML profile for CoCons can only be developed in future research if the CoCon concepts can be covered

with current UML mechanisms and semantics. New metatypes are suggested in figure 2. Usually, this would go beyond a true 'profile' of UML. However, the new metatype 'ContextBasedConstraint' only *specializes* the 'Constraint' metatype. 'ContextCondition' is a utility class that only illustrate the difference between the existing metatypes and the new one. Thus, the integration suggested is based on standard UML concepts and refines them in the spirit of UML profiles. It falls into the category of lightweight formal methods.

In the winter semester 2001/02 a 'CCL-plugin' for the open source UML editor 'ArgoUML' wasimplemented at the TU Berlin in order to specify and to automatically protect CoCon specifications during modeling. It is available at `ccl-plugin.berlios.de` and demonstrates how the standard XMI format for representing UML in XML must not be changed in order to save or load models containing CoCons. Hence, CoCons can be integrated into UML without modifying the standard.

3.7 Comparing OCL to Context-Based Constraints

According to [10, 15], three kinds of constraints exist: preconditions, postconditions and invariants. Typically, the *Object Constraint Language OCL* summarized in [16] 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 selects 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' [5] 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_1 level because this paper focuses on using them during design. For example, if a CoCon states that "*package 'X' must contain all classes belonging to the operational area 'field service'* ", then the model should be checked for whether it violates this CoCon already during design. Using OCL, the designer may create a stereotype ≪contains-all-classes-belonging-to-the-field-service≫ and assign a constraint to this stereotype on the M2 level. As discussed before, there is no formal constraint language for selecting a model element due to its metadata. Hence, the constraint must be written down in natural language and cannot be verified automatically. Even if OCL constraints could iterate over all model elements in all diagrams and select those fulfilling the context condition, modifying the *meta*model each time the requirements change is not appropriate.

There used to be a lot of interest in machine-processed records of *design rationale*. According to [11], the idea was that designers would not only record the results of their design thinking, but also the reasons behind their decision. Thus, they would also record their justification for why it is as it is. CoCons record design decisions that can be automatically checked. They represent certain relevant requirements in the model. The problem is that designers simply don't like writing down design decisions. The challenge is to make the effort of recording the rationale worthwhile and not too tedious for the designer. As a reward for writing down essential design decisions via CoCons they reap the benefits summarized in section 4.3.

4 Conclusion

4.1 Applying CoCons in the Development Process

In this section the application of CoCons throughout the software development process is sketched. **During requirements analysis** the business experts must be asked specific questions in order to find out useful context properties and CoCons. They may be asked about which business exist rules for which context. Examples: "*Which important workflows exist in your business? And for each workflow, which business objects belong to this workflow and which requirements belong to this workflow*". Then it is possible to state that all business objects belonging to workflow 'X' must comply with requirement 'Y'. Currently a CoCon-aware method for requirements analysis is being developed at the Technical University of Berlin in cooperation with Eurocontrol, Paris.

The benefits of considering both requirements and architecture when **modeling** a system are discussed in [12]. The application of CoCons during modeling 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. This paper cannot discuss how to verify or 'proof' CoCon specifications *automatically* because for each CoCon type and for each abstraction level in the development process different requirement verification mechanisms

are necessary. Please refer to the verification of CoCons integrated into the Design Critiques ([14]) mechanism of ArgoUML via the CCL Plugin.

During deployment a CoCon-aware configuration file checker can automatically protect requirements. Likewise, the notion of contextual diagrams is introduced in [7] in order to cope with the intrinsic complexity of configuration knowledge. A deployment descriptor checker for Enterprise Java Beans is currently being developed at the TU Berlin.

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. 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 autarkic component and communication with this component can be monitored *externally* for whether it complies with the CoCon specification **at runtime**. A prototypical framework is currently beeing integrated into an application server at the TU of Berlin in cooperation with BEA Systems and the Fraunhofer ISST. Thus, legacy components or 'off the shelf' components can be forced to comply with new requirements.

4.2 Limitations of Context-Based Constraints

Taking only the tagged values of a model element into consideration bears some risks. It must be ensured that theses values are always up-to date. Whoever holds the responsibility for the values must be trustworthy. Confidence can be assisted with encryption techniques. Within one system, only one ontology should be used. For instance, the workflow 'New Customer' must have exactly this name in every part of the system, even if different companies manufacture its parts. Otherwise, string matching gets complex when checking a context condition.

Context properties are highly abstract and ignore many details. For instance, this paper disregards the dependencies between context property values. Handling dependent context property values is explained in [2]

4.3 Benefits of Context-Based Constraints

In contrast to grouping techniques, e.g. packages or stereotypes, context properties facilitate handling of overlapping or varying groups of model elements that share a context even across different model element types or diagrams. Hence, one requirement referring to several, possibly unassociated model elements can now be expressed via one constraint. Context properties allow *subject-specific, problem-oriented views* to be concentrated on. For instance, only those model elements belonging to workflow 'X' may be of interest in a design decision. Many concepts for specifying metadata exist and can be used instead, if they enable a constraint to select the constrained elements via their metadata.

Decision making is an essential activity performed by software architects in designing software systems. The resulting design must satisfy the requirements

while not violating constraints imposed on the problem domain and implementation technologies. However, in complex domains, no one architect has all the knowledge needed to make a complex design. Instead, most complex systems are designed by teams of stakeholders providing some of the needed knowledge and their own goals and priorities. The 'thin spread of application domain knowledge' has been identified by [6] as a general problem in software development. In complex domains even experienced architects need knowledge support. For instance, they need to be reminded which of the requirements apply to which part of the system. The model should serve as a document understood by designers, programmers and customers. CoCons can be specified in easily comprehensible, straightforward language. They enforce a system's compliance with requirements. Even the person who specifies a requirement via CoCons must not have complete knowledge of the system due to the indirect association of CoCons to the system parts involved. CoCons associate relevant requirements with related elements of the system's model.

In software engineering, it has long been recognized that inconsistency is a fact of life. Evolving descriptions of software artefacts are frequently inconsistent, and tolerating this inconsistency is important if flexible collaborative working is to be supported. The abstract meta-information belonging to a model element can be ascertained out is an early lifecycle activity. When identifying the context property values the model element must not be specified in full detail. Metadata can supplement missing data based on experience or estimates.

Maintenance is a key issue in *continuous software engineering*. CoCons help to ensure consistency during system evolution. A context-based constraint serves as an invariant and thus prevents the violation of design decisions during later modifications of UML diagrams. It assists in detecting when design or context modifications compromise intended functionality. It helps to prevent unanticipated side effects during redesign and it supports collaborative design management. The only constant in life is change, and requirements tend to change quite often. This paper suggests improving the adaptability of a system model by enforcing conformity with meta-information. This meta-information can be easily adapted, whenever the context of a model element changes. In this case, some CoCon specifications may apply anew to this model element, while others may cease to apply. Furthermore, the CoCon specifications themselves can also be modified if requirements change. Each deleted, modified or additional CoCon can be automatically enforced and any resulting conflicts can be identified as discussed in [2]. 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 – they improve the traceability of contexts and requirements. CoCons can be verified during modeling, during deployment and at runtime. They facilitate description, comprehension and reasoning at different levels and support checking the compliance of a system with requirements automatically. According to [9], automated support for software evolution is central to solving some very important technical problems in current day software engineering.

References

[1] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In B.Rumpe and R.B.France, editors, *2nd International Conference on the Unified Modeling Language, Colorado, USA*, volume 1723 of *LNCS*, pages 249–264. Springer, 1999.

[2] Felix Bübl. The context-based component constraint language CCL. Technical report, Technical University Berlin, available at `http://www.CoCons.org`, 2002.

[3] Felix Bübl and Andreas Leicher. Desiging distributed component-based systems with DCL. In 7^{th} *IEEE Intern. Conference on Engineering of Complex Computer Systems ICECCS, Skövde, Sweden*. IEEE Computer Soc. Press, June 2001.

[4] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[5] 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.

[6] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Comm. ACM*, 31(11):1268–1287, 1988.

[7] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Contextual diagrams as structuring mechanism for designing configuration knowledge bases in UML. In A. Evans, S. Kent, and B. Selic, editors, *3rd International Conference on the Unified Modeling Language, York, United Kingdom*, volume 1939 of *LNCS*. Springer, 2000.

[8] Robert W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 5(6):345, 1962.

[9] Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

[10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[11] Thomas P. Moran and John M. Carroll, editors. *Design Rationale : Concepts, Techniques, and Use (Computers, Cognition, and Work)*. Lawrence Erlbaum Associates, Inc., 1996.

[12] 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.

[13] OMG. UML specification v1.4 (ad/01-02-14), 2001.

[14] Jason E. Robbins and David F. Redmiles. Software architecture critics in the argo design environment. *Knowledge-Based Systems. Special issue: The Best of IUI'98*, 5(1):47–60, 1998.

[15] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Reading, 1997.

[16] Jos B. Warmer and Anneke G. Kleppe. *Object Constraint Language – Precise modeling with UML*. Addison-Wesley, Reading, 1999.

[17] Stephan Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[18] 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.