Keeping Track of Crosscutting Requirements in UML Models via Context-Based Constraints

Felix Bübl
Epigenomics AG
Berlin, Germany
fbuebl@cocons.org

ABSTRACT

One crosscutting requirement (also called aspect) affects several parts of a software system. Handling aspects is well understood at source-code level or at runtime. However, only a few aspect-oriented approaches handle other software artefact types, like UML models, configuration files, or database schema definitions. Instead of re-writing the same aspect newly for each artefact type, this paper suggests to write down aspects independent of artefact types.

But, wait a minute: Where do we weave in an aspect if its pointcut doesn't refer to artefact details? Which places does such an aspect affect? This paper suggests expressing aspects via Context-Based Constraints (CoCons). They select their constrained system elements according to the element's context. For instance, CoCons affect all system elements used in a certain department, workflow, or location. CoCons are easy to grasp for users and customers because they express business requirements without referring to technical details. This paper focuses on how to express crosscutting requirements in UML models via CoCons and compares CoCons to the Object Constraint Language OCL.

1. INTRODUCTION

1.1 It's a Long Way to the Top if you wanna Rock'n'Roll

This article has already been submitted to several conferences without success. The reviewers of these previous conferences had good reasons for rejecting this article. For example:

"The paper has no clear focus and is badly written."

Thanks to the bags of critics, the maturity of the article has hopefully improved in the meantime. For instance, it doesn't claim anymore that Earth is a flat disk. Well, we still does not learn grammar at school, does we? On the next pages, you will find several quotations from reviewers that helped improving this article, like the reviewer comment above.

1.2 Privacy Policy Example

Before going into details, we start with two examples for crosscutting requirements where one requirement affects several system elements.

Let's assume that a component-based software system should comply with the following privacy policy:

> All components handling personal data must be inaccessible to all components used in the workflow 'Create XYZ Report'

A software system consists of different artefact types, like models, source code, or configuration files. Which elements of which system artefact should be checked for whether they do (or do not) access which other elements in order to meet the privacy policy?

1.3 Availability Requirement Example

Moreover, let's assume that our distributed software should meet the following availability requirement:

> All data needed in the workflow 'Create XYZ Report' must be allocated to all computers belonging to the 'Sales' department.

Due to this requirement, any computer of the sales department can access all the data needed in the workflow 'Create XYZ Report' even if the network fails. Hence, the availability of the workflow 'Create XYZ Report' is ensured on these computers. But, which data should be checked if it is (not) allocated to which computers? Before providing the answer, the next sections outline the problems we want to solve.

1.4 Goal: Identify Join Points Independent of Artefact Types

According to [21], **aspect-oriented programming languages** supplement programming languages with crosscutting concerns that are called aspects. The aspects are developed separately from the normal source code and are weaved into the source code on compile time or even dynamically at runtime. A place where to weave in an aspect is called **join point** in AspectJ ([3]). A **pointcut** defines the conditions under which to weave in an aspect – it defines a query for selecting the join points. According to [35], defining the pointcuts is independent design issue and can be accomplished separate from other tasks such as modelling the crosscutting effect (which is called **advice** in AspectJ).

Typically, a pointcut selects join points by referring to source code details like names of classes or methods. But, stakeholders who don't know anything about the source code should be able to understand and agree with crosscutting requirements. Hence, our goal is to define pointcuts that work without knowing the source code at all. For example, we want to determine where to control the privacy policy (see section 1.2) without knowing any technical detail about the system's components.

In a system model, one crosscutting requirement relates to several model elements that may not be associated with each other or may even belong to different diagrams. In source code files, one crosscutting requirement can be reflected in many different lines of code. Likewise, one crosscutting requirement can affect several places in configuration files. At runtime, one crosscutting requirement can be realized in several binary components that may not invoke each other or may even run on different platforms.

It takes too much effort if the same requirement must be stated newly for each software system artefact at each affected place. On the contrary, we try to express crosscutting requirements independent of the modelling language, the programming language, or the platform in order to apply the same requirement expression to all these artefact types. In order to achieve this goal, we need to express pointcuts without referring to system artefact details. For example, the privacy policy (see section 1.2) should be written down only once in an abstract, artefact-type-independent manner that can be reflected in each software system artefact at each affected place. In this paper, we focus on UML models. Our approach, however, can be adapted to other artefact types, too.

1.5 Goal: Detect Violated or Contradicting Crosscutting Requirements

Contradicting or violated requirements should be detected as soon as possible. We try to express crosscutting requirements in a way that enable tools to find both violated and contradicting requirements automatically. For example, software tools should be able us to detect any system artefact element that violates the privacy policy, and they should detect if other crosscutting requirements contradict the privacy policy. In order to enable tools to understand our crosscutting requirements, we specify them as constraints.

1.6 The Approach in Brief

Specifying crosscutting requirements via context-based constraints (CoCons) is explained in [8] on a formal, abstract level. This paper compares them with aspects, applies them to UML models, and compares them with OCL. Their basic notion can be explained in just a few sentences:

1. We annotate the system artefact elements with formatted metadata called 'context properties'. A context property describes its element's context. As explained in section 2.1, context is any information that can be used to characterize the situation of an element.
2. A CoCon expresses a condition on how system elements must (or must not) relate to each other.
3. A CoCon selects its constrained elements (= join points) according to their context properties – the pointcuts of a CoCon are determined by the context of the constrained elements.

For instance, we could use context properties to mark each element that handles personal data, and to mark each element that is used in the 'Create XYZ Report' workflow. Then, the privacy policy in section 1.2 can be expressed via the following CoCon:

> All elements that handle personal data must be inaccessible to those elements that are used in the Create XYZ Report workflow.

1.7 Section Overview

Section 1 has introduced two example scenarios, outlined the problem we want to tackle here, and explained the basic terms of aspect-oriented software development.

The next two sections suggest a solution approach. Section 2 presents the notion of context used here, and section 3 describes context-based constraints (CoCons).

Theoretically, CoCons can be applied to any software system artefact type. In this paper, however, we focus on one artefact type only: UML models. The standard constraint language for UML is OCL. Hence, section 4 compares CoCons and OCL. Afterwards, section 5 suggests how to apply CoCons to UML models. Subsequently, section 6 discusses related research on crosscutting requirements engineering. These three sections contain the main contribution of this article. Only by going into details for one artefact type, we can explain why and how this paper suggests new notion of constraints. Finally, section 7 discusses the limitations and benefits of the approach.

> "Most of the paper is about comparisons. This is what should be the content of a related work section (which is missing in this paper)."

Well, related literature is scattered all over the paper in a crosscutting way because different sections focus on different research areas: related research on aspects is discussed in a different section than related research on UML.

2. INTRODUCING CONTEXT PROPERTIES

2.1 What is Context?

Each software system element resides in an infinite number of contexts – according to [34, 19], it is impossible to list all contexts of an element because it is not possible to completely define what an element denotes. All context definitions developed in computer science fail to provide a *general* theory of context as discussed in [17]. Only limited context models can be handled. Thus, we stick to a simple and limited context model:

- The context of an element characterizes the situation(s) in which the element resides as defined in [15].
- The context of a context is ignored here.
- Context that is not part of or managed by the system can be taken into account.

This notion needs a precise definition of 'situation'. In situation calculus ([14]), **situation** is defined as structured part of the reality that an agent manages to pick out and/or to individuate. This definition suits well for this paper because context is used here for distinguishing those elements that are involved in a requirement from the other elements. A context is not a situation, for a situation (of situation calculus) is the complete state of the world at a given instant. A single context, however, is necessarily partial and approximate. It cannot *completely* define the situations. In contrast, it only characterizes the situation.

> "One of the important concepts that is used several times in this paper is context. However, the concept context was not clearly defined."

As explained at the start of this section and in [17], context remains a spurious concept. We can only arrive at a clear definition if we focus one application. Before providing context examples in the next section that hopefully clarify the definition of context used here, this section explains why we do **not** use the typical notion of context in software engineering.

A software system consists of artefacts, like source code files, configuration files, or models. One artefact can consist of several elements. An **internal element** is contained in at least one of the system's artefacts. For example, a component, a method, or a method's parameter are internal elements because they are defined within the system's artefacts. On the contrary, an **external context** is not contained in any of the system's artefacts. Most context definitions used in software engineering only consider internal context of a software system element: internal context only refers to other internal elements. It does not refer to external elements. For instance, the 'context of a component' is defined as 'the required interfaces and the acceptable execution platforms' of components in [36]. This is an internal notion of context because it only refers to internal elements that are part of the system: other components or containers are defined as context of a component. In the UML 2.0 specification ([25]), context is defined as 'a view of a set of related modelling elements for a particular purpose, such as specifying an operation'. Again, this is an internal notion of context: the context of a model element refers to other internal model elements.

On the contrary, this paper proposes also to take external contexts of the system into account. The users and customers understand the external context of a system better than internal technical details. The next section suggests how to express external context and gives examples.

2.2 Context Properties: Formatted Metadata Describing Elements

The context of a software system element can be expressed as metadata. The attribute-value model is commonly used as metadata format today. As well, we suggest expressing context in the simple attribute-value syntax: A **context property** consists of a name and a set of values. Some examples are provided:

- The values of the context property 'Workflow' reflect the names of the workflows in which the associated element is used.
- The value 'Yes' or 'No' of the context property 'Personal Data' signals whether an element handles data of private nature.
- The values of the context property 'Operational Area' tell, in which department(s), module(s), or domain(s) the associated element is used. They provide an organisational perspective. For instance, 'Sales', 'Human Resources', 'Controlling', and 'IT' are values of the context property 'Operational Area'.

Such context information is typically not part of a system's source code. Still, we need to store it somewhere if we want to refer to it. In order to enrich a system with context information, we don't have to modify its source code or its binary components. On the contrary, we can manage the context-information in an external repository. Of course, the context properties can also be managed in the source code. For instance, the Java metadata facility, a part of J2SE 5.0, is a significant recent addition to the Java language. It includes a mechanism for adding custom annotations to your Java code, as well as providing a programmatic access to metadata annotation through reflection.

A context property groups artefact elements that share a context. Object-oriented grouping mechanisms like inheritance, stereotypes or packages are not used because the values of a context property associated with one element might vary in different configurations or even change at runtime. An element is not supposed to change its stereotype or its package at runtime. Context properties facilitate handling crosscutting requirements because they are a simple mechanism for grouping otherwise possibly unassociated model elements - even across different views, artefact types, or platforms.

3. INTRODUCING CONTEXT-BASED CONSTRAINTS (COCONS)

This section describes a new technique for specifying crosscutting requirements called 'CoCons'.

3.1 Intuitive Definition of Context-Based Constraints

A **context-based constraint** (CoCon) expresses a condition on how its constrained elements must relate to each other. This condition $C(x, y)$ is called **CoCon-predicate** here. Different CoCon-predicates exist. For example, a CoCon-predicate can express that 'x must (or must not) be accessible to y' (as in the security requirement in section 1.2). Another CoCon-predicate can express that certain elements must (or must not) be allocated to certain computers (distribution requirement, see section 1.3). In terms of AspectJ ([3]), $C(x, y)$ expresses an advice.

> "You tell us that $C(x, y)$ expresses an advice. I'd like to see an explanation for this. Doesn't it express a constraint?"

So, what's the difference between constraints and advices? In UML ([25]), a constraint is a semantic condition or restriction expressed which must be true for the model to be well formed. In AspectJ, advice is the implementation of behaviour that crosscuts the set of execution points defined by a pointcut. An advice can implement any kind of behaviour. For instance, an advice can implement a constraint: it can either monitor if the system meets the constraint's condition in each join point, or it can enforce the constraint's condition (if the system doesn't meet the constraint's condition at a specific join point then change the system so that it meets the constraint's condition). A CoCon cannot express every possible advice. It can only express a condition. This condition $C(x,y)$ can either me monitored or enforced by an advice. This paper focuses only on those aspects/advices that monitor or enforce constraints. We will continue comparing aspects and CoCons in section 6.

A CoCon could relate many *sets* of constrained elements. In this paper, we only discuss CoCons that relate *two* sets of elements (one is called target set, and the other one scope set) as expressed via the following predicate logic formula:

$$\forall x,y : TCC(x) \wedge SCC(y) \rightarrow C(x,y)$$

The variable x holds all elements in the target set, and the variable y hold all elements in the scope set. The predicate $C(x,y)$ on the right side of the formula is a called CoCon-predicate because it defines the semantics of the CoCon. For example, $C(x,y)$ can represent `x MUST BE ACCESSIBLE TO y`, or it can represent `x MUST BE LOGGED WHEN CALLING y`. $C(x,y)$ is a binary relation – it expresses how x relates to y.

Besides $C(x,y)$, all the other predicates refer to a different level: they define *which* x must relate to *which* y. The predicates $TCC(x)$ and $SCC(y)$ define context conditions.

"What is TCC and what is SCC?"

TCC is the abbreviation of Target Set Context Condition, an SCC stands for Scope Set Context Condition. A **context condition** selects artefact elements according to their context property values. It is a point cut condition which only refers to context properties of system elements as discussed later on in section 6.1. Different query languages exist for expressing a context condition - the right choice depends on in which format the context metadata is stored. If the context properties of each artefact element are stored in a relational schema then a relational query language, e.g. SQL, can be used to express context conditions. If the context properties are stored in a hierarchical XML schema then a query language for XML documents can be used, e.g. XQuery.

The Context-Based Constraint Language CCL has been defined in [7]. The syntax of CCL is not explained here, because it resembles plain English and is easily understood. For example, the privacy policy described in section 1.2 can be expressed in CCL as follows:

```
ALL COMPONENTS WHERE ‘Personal Data’ CONTAINS
‘True’ MUST NOT BE ACCESSIBLE TO ALL COMPONENTS
WHERE ‘Workflow’ CONTAINS ‘Create XYZ Report’
```

3.2 Two-Step Approach for Defining CoCon-Predicate Semantics

CoCons can be applied to *artefact types* at different development levels, like *requirement specifications* at analysis level, *UML models* at design level, *Java files* at source code level, or *component instances* at runtime. Figure 1 illustrates the two-step approach for defining semantics of CoCon-predicates:

- The **artefact-type-independent semantics** of a CoCon-predicate do not refer to specific properties of an individual artefact type. For instance, the artefact-type-independent semantics of an `ACCESSIBLE TO` CoCon are defined in plain English as: its target set elements are accessible to its scope set elements.
- The **artefact-type-specific semantics** of a CoCon-predicate define how to check artefacts of a certain type whether the artefact element x relates to the artefact element y as demanded by the CoCon-predicate. For example, how can we check a UML model if its elements comply with the CoCon-predicate 'x must be accessible to y'? The artefact-type-specific semantics definition of `ACCESSIBLE TO` CoCons for UML models can be defined via the object constraint language OCL as discussed in section 4.

"A simpler to that problem would be to describe the crosscutting requirement in design and then automatically translate it into code."

This comment reflects the traditional software engineering thinking. It's a great vision to specify requirements in design and then translate them into code, but it won't work if the system design artefacts are incomplete, outdated, or don't exist at all. The traditional approach is not simpler because it demands an up-to-date, complete, and hopefully formal model (called 'master-model' from now on).

In contrast, the two-step semantics approach works for specific artefact types regardless of which other system artefacts exists, are outdated, or don't exist. For instance, in order to check if a distributed system meets the availability constraint of section 1.3 via traditional software engineering, we build a distribution model (for example a UML deployment diagram even though UML isn't a formal language), check it, and then generate the JBoss configuration files from this distribution model. In the two-step approach, however, it is sufficient to check the JBoss files - we don't need a UML deployment diagram. If a UML model of a system exists, then it can be checked for compliance with CoCons via the UML-specific semantics of these CoCons. If JBoss configuration files exist, then you need the JBoss-specific semantics of these CoCons. We don't need a master-model that contains all details needed to check the constraint. On the contrary, we directly check those artefacts that express the relevant details. It's still important to document as many details of your system as possible, but sometimes we need to check system artefacts without having a formal master-model for them.

"The author says in that he uses a two-step approach for defining semantics, he rather defines two kinds of semantics."

Oops. Actually, this comment reveals a weakness of the two-step approach. According to its definition at the beginning of this section, the artefact-independent semantics can be expressed in plain English. They don't need to be formally defined (which would again call for a master model on which they are formally defined). But: plain English language can easily be misunderstood. Nevertheless, as long as the plain English semantics are not ambiguous,

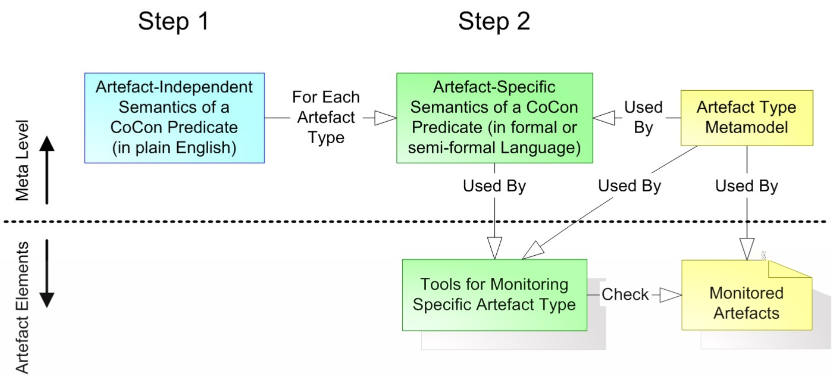
Step 1
Step 2
Artefact-Independent Semantics of a CoCon Predicate (in plain English)
For Each Artefact Type
Artefact-Specific Semantics of a CoCon Predicate (in formal or semi-formal Language)
Used By
Artefact Type Metamodel
Used By
Used By
Used By
Meta Level
Artefact Elements
Tools for Monitoring Specific Artefact Type
Check
Monitored Artefacts

Figure 1: Two-Step Approach for Defining the Semantics of CoCon-Predicates

we can do without formal semantics or a master-model. It's not perfect, but achievable.

> "You claim that your approach is independent of artifact types. To what extent is this really true, or example, does your approach work with artifacts that are plain text documents?"

The two-step semantics approach only works for artefact types, which stick to a defined syntax and semantics (I tried to avoid the term 'semi-formal' artefact types). Hence, it doesn't work for plain text document, because we cannot define bulletproof semantics for English.

3.3 Which Requirements can be Expressed via CoCons?

All in all, 22 CoCon-predicates for component-based systems are defined in [7]. They address concerns like 'x must be logged when calling y' which are typically realized via aspect-oriented programming. Future research hopefully will examine additional CoCon-predicates. These currently 22 CoCon types are grouped in five different families of CoCons:

- Access Permission CoCons express which components *must (or must not) be accessible to* which other components.
- Communication CoCons control whether a method call between components *must be (or must not be)* intercepted in order to handle it. For instance, `x MUST BE LOGGED WHEN CALLING y` or `x MUST BE ASYCHRONOUSLY CALLING y` express crosscutting communication concerns.
- Distribution CoCons express which components *must (or must not) be allocated to* which computers. For example, `x MUST BE ALLOCATED TO y` or `x MUST BE REPLICATED TO y` express crosscutting distribution requirements.
- Information-Need CoCons express which users *must (or must not) be notified of* which documents at runtime. For instance, `x MUST BE NOTFIED OF y` is a crosscutting information need concern.
- Inter-Value CoCons express whether elements in a certain context must (or must not) reside in another context. For example, `x MUST RESIDE IN THE SAME CONTEXT AS y` is a crosscutting context-property-value concern.

> "Why are there 22 CoCon predicates? Aren't there arbitrary many? And why are there only 5 families of CoCons? How did the author find them?"

These 22 CoCon predicates were identified in several collaborations outlined in section 5.3. But of course, additional crosscutting concerns can be expressed via CoCons as well, but they haven't been examined yet.

> "The authors fail to discuss how reasonable or how feasible it would be to write all requirements as CoCons."

Is it not possible to write down all requirements via CoCons. A context condition can be restricted to only select elements of a specific element type. For instance, it can be restricted to select nothing but components, or nothing but classes. A CoCon-predicate expresses a condition on how two elements must relate. The expressiveness of a CoCon-predicate depends on the range of (= the element type in focus of) its context conditions. All elements of one element type share their type's properties. For example, all elements having the element type 'component' share the type property 'a component can have interfaces'. In UML, these type properties are defined in the metamodel. But, not all components have the same interface. Therefore, a CoCon restricted to components cannot express conditions on specific properties of one interface of one of its constrained components. Instead, a CoCon only can express conditions on the type's properties of those element types to which its context conditions are restricted.

As soon as the context property values change or elements are added or removed, the same CoCon can apply to other elements that are *unknown* when specifying the CoCon. Besides its type properties, all other properties of the constrained element are unknown.

Up to now, we did not examine CoCon-predicates $C(x)$ which focus on a single type property, as in the condition 'the attribute age of a customer must have a value greater then 17'. Many other constraint languages exist for expressing conditions on single properties. On the contrary, we focused on CoCon-predicates expressing how elements having certain type properties *relate to* each other .

4. COMPARING CONTEXT-BASED CONSTRAINTS AND OCL

Many techniques exist to date that augment UML with properties and constraints. This section explains why CoCons add a new notion of constraints, which is not covered by other approaches. Typically, the *Object Constraint Language OCL* ([38]) is used for the constraint specification of UML models. This section explains how to express CoCons via OCL and discusses the differences. It does not claim that CoCons are superior to OCL.

"The paper proposes a more abstract constrained based language that could replace OCL."

No. CoCons cannot replace OCL.

"The authors claim that their approach provides a richer constraint language than the standard OCL."

No. CoCons are not richer than OCL. Instead, they add a different abstraction layer.

4.1 UML-Specific Semantics of `ACCESSIBLE TO` CoCons

This section discusses the translation of `ACCESSIBLE TO` CoCons into OCL via the privacy policy example described in section 1.2 and expressed in CCL at the end of section 3.1. This CCL expression is two lines long. It can incompletely be specified in OCL as:

```
context component inv: self.taggedvalue
    ->select(tv | tv.dataValue = "Create XYZ
    Report") .type -> select(td | td.name =
    "Workflow") -> notEmpty()

implies self.clientDependency.supplier
    -> select(i | i.oclIsTypeOf(Interface))
    .clientDependency
    -> select(d | d.oclIsKindOf(Abstraction)
    and d.stereotype.name = "realize"
    and d.supplier.oclIsKindOf(Classifier))
    .supplier -> select(c |
    c.oclIsTypeOf(Component)) .taggedvalue
    newline -> select(tv | tv.dataValue = "True")
    .type -> select(td | td.name = "Personal
    Data") -> Empty()
```

This OCL expression states that a component having the tagged value 'Workflow: Create XYZ Report' must not (= '`Empty()`' at the end of the OCL expression above) depend on the interface of a component having the tagged value 'Personal Data: True'. The violation of this OCL expression is illustrated in the UML deployment diagram shown in figure 2. In this diagram, the dependency relationship (represented as dotted arrow in the diagram, and as `clientDependency` in the OCL expression) specifies that component 'A' invokes component 'B'. However, this invocation violates the privacy policy due to the context property values of A and B.

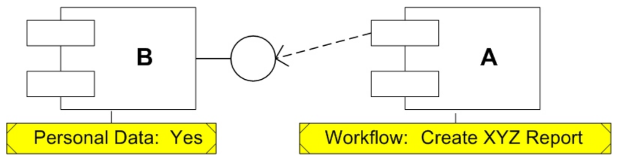

Figure 2: The component 'A' invokes the component 'B' and, thus, violates the privacy policy of section 1.2

Why should anyone use the new language CCL for expressing CoCons at all if the prevailing (standard!) language OCL already can express the same privacy policy? The problem is that the OCL expression above is incomplete - it needs to be much longer to completely map the CCL expression to UML. It would be much longer if it would cover the following missing details.

First, it only defines inaccessibility for component *types*, but not for component instances. A UML model, however, can contain both component types and component instances. Therefore, additional OCL expressions are needed which consider the component instances.

Moreover, the OCL expression above does not consider communication between components in various behavioural diagrams, such as activity diagrams, sequence diagrams, and state machine diagrams, communication diagrams, or interaction overview diagrams. For each of these diagrams, the concept of (in)accessibility both between component types *and* between component instances must also be considered in the specification of artefact-specific semantics. Again, additional OCL expressions are needed which consider communication via one of these metaclasses: Action, Activity, Behavior, CommunicationPath, Connector, ControlFlow, InformationFlow, Interaction, Message, ObjectFlow, Operation, Reception, Signal, or Stimulus.

In addition to 'plain' standard UML, some component specification approaches consider *composition* of components. The OCL expression given above does not consider composition (or aggregation, or PackagableElement): if the component 'B' in figure 2 does not manage personal data, but 'B' is composed of other components among whom at least one component handles personal data, than 'A' must not invoke an operation of an interface of 'B'. Furthermore, the OCL expression given above does not handle recursion (a solution is described in [12]): 'A' must not invoke an operation of an interface of 'B' if 'B' calls another component 'C' handling personal data in order to execute A's call. Again, some additional OCL expressions are needed which consider composition and recursion.

Moreover, an artefact using additional concepts which are not part of the artefact's standard needs special artefact-specific semantics: the OCL translation of the CoCon example given above must be adapted if any profile is used that adds a new notion of (in)accessibility to UML. For example, the 'UML Components' approach introduced in [9] specifies components via stereotyped classes. The incomplete OCL expression given above does not consider stereotyped classes (neither in component nor in deployment nor in sequence or collaboration diagrams). In 'UML Components', stereotyped classes represent component *types*. Thus, component *instances* are represented as objects in 'UML Components'. If we use a profile, we need several additional OCL expressions that consider the profile's semantics.

In case of `ACCESSIBLE TO` CoCons, the artefact-type-*independent* semantics definition in plain English consists of three words: 'is accessible to'. On the contrary, the corresponding listing of artefact-type-specific OCL expressions for expressing the precise semantics for UML models is about as long as this article because it considers a lot more details. CoCons stay on the artefact-type-independent, abstract level. OCL, however, is too close to programming for expressing requirements at this abstraction level. The effort of writing down a requirement in the minutest details is unsuitable if the details are not important. The designer can ignore many details by expressing the same requirement via the Context-Based Constraint Language CCL instead of OCL. Moreover, it is easier to adapt the short artefact-type-independent CCL expression instead of changing all the OCL expressions if the corresponding requirement changes.

> "Your claim that CoCons are much simpler than corresponding OCL constraints is a self-deception: in your approach, the complexity is hidden in the precise definition of the semantics of basic CoCon statements such as ACCESSIBLE TO."

That's correct. But this self-deception enables us to discuss `ACCESSIBLE TO` constraints with customers and users. This section has outlined how many details must be considered when defining the UML-specific semantics of x `is ACCESSIBLE TO` y. But, we need to define the UML-specific semantics only once (at least until a new version of UML is released) and then 'hide' the complexity by only using the artefact-independent expression. Besides the detail level, two other differences between CoCons and OCL are discussed in the next two sections.

4.2 CoCons can be Verified Already at the Same Meta-Level

The Object Management Group (OMG) describes four metal-levels: 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). Note that level M_{i-1} elements are *instances* of level M_i elements.

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' ([13]) of an invariant is an *instance* of the associated model element. In order to check a M_1 level OCL constraint, either M_0 level instances must be simulated as proposed in [31], or the OCL constraint must be translated into M_0 level source code which evaluates the constraint at runtime as suggested in [18]. In order to check the compliance of a model element (M_1 level) with an OCL constraint, the OCL constraint must be specified on metamodel level (M_2). Hence, the OCL expressions defining the UML-specific semantics for `ACCESSIBLE TO` CoCons described in the previous section 4.1 must be specified on the *metamodel* level. As long as we use OCL to express context-based constraints, we need to modifying the *meta*model each time the requirements change - this is simply not appropriate.

On the contrary, M_1 level model elements can already be checked for compliance with CoCons expressed on M_1 level as explained next.A CoCon's context condition can be evaluated as soon as the corresponding context property values are defined in the UML model via tagged values on M_1 level.. Hence, the model elements constrained by a M_1 level CoCon can already be identified on M_1 level. Additionally, each pair of related constrained elements must be checked if it fulfils the CoCon-predicate $C(x, y)$. The OCL semantics definition of `ACCESSIBLE TO` provided in the previous section are be expressed on M_2 level and refer to M_1 level model elements. Therefore, an `ACCESSIBLE TO` CoCon expressed on M_1 level can be checked on M_1 level. This is a major difference between CoCons and the prevailing notion of constraints.

> "The author claims that CoCons can be verified at the same meta-level. It is not clear how this can be done."

When we express a CoCon on M_1 level (in a UML model), we need to evaluate its context conditions and its CoCon-predicate in order to verity the CoCon. Both checks work on the same metalevel M_1 for the following reasons.

Evaluate a Context Condition: The context properties of the model elements are defined on M_1 level (as tagged values in the UML model). In order to evaluate the CoCon's context conditions, we only need the M_1 model because it contains both the model elements and their context properties. We neither need to refer to any M_0 or M_2 information to select the constrained elements.

Evaluate a CoCon-Predicate: As explained in the previous section 4.1, the UML-specific semantics of a CoCon-Predicate are defined in OCL at *metamodel* level (M_2). These M_2 OCL constraints can be verified at M_1 level because OCL constraints can be verified on the next lower metalevel as explained in the second paragraph of this section.

According to [16], those persons that produce traceability links - mainly the members of the development team - have different goals and priorities than the users - mainly the clients, managers, and the test and maintenance team – who use these traceability links in order to check whether the system complies with the requirements. According to [26], the designers and developers simply do not see the benefits that may accrue to the final product compared to the time and effort required for producing the traceability links. CoCons enable the designers and developers to instantly see the benefits because a model element can be checked for compliance with requirements at the moment when the relevant context property values are associated with it. A tool can immediately warn designers if their model violates a CoCon. This direct feedback can encourage them in associating model elements with context property values.

> "The paper does not really deal with tracing requirements."

According to [27], applying requirements traceability to a software system starts with the following two tasks:

1. **Traces Definition**: It must be defined what (kinds of) objects should be traced and what (kinds of) traceability links are needed between those objects.
2. **Traces Production**: The traces are recorded by associating traceability links with the relevant objects.

A **traceability link** is expressed by associating meta information with an element. According to [29], the following kinds of traceability links exist:

1. A **satisfaction link** is defined between an element that represents a constraint or goal, and another element that satisfies it.
2. A **dependency link** is defined between an element whose modification will impact another element.
3. An **evolution link** is defined between an element that acts as a replacement of another one, such that only the latter is still valid.
4. A **rationale link** is defined between an element and an explanation of this element.

Context properties refine the notion of rationale links: a **context link** is defined between an element and its context by associating the element with context property values. This context link is a rationale link because it explains the element. A CoCon is a satisfaction link because it expresses a condition on how its constrained elements must relate to each other. Up to now, satisfaction links tell which goals the element must fulfil to which they are associated. On the contrary, a CoCon must not be directly associated with its constrained elements. Instead, context property values are directly associated with the elements. They do not tell which goals the element must fulfil with which they are associated. Instead, they tell the context of their element. A CoCon can express a goal by referring to the element context. Hence, CoCons are **indirect satisfaction links** — they provide a new notion of requirements traceability.

Higher-level requirements must be decomposed to a more refined level in order to provide a link from initial requirements to actual system elements that satisfy those requirements. During this recursive decomposition process, low-level requirements are *derived* from higher-level requirements. Both original an derived requirements are *allocated to* system elements. According to [29], an **requirements allocation table** is the common mechanism used to maintain this information. However, keeping track of each individual requirement or element becomes more and more difficult if the number of requirements or elements grows. Furthermore, this difficulty increases if the requirements or elements change frequently. In case of large-scale or frequently changing systems, it takes much effort to maintain an requirements allocation table that directly links requirements to individual elements. Instead, CoCons enable to specify requirements for possibly large groups of elements. They allow for indirect, *adaptive* selection of all the elements involved in the requirement.

4.3 CoCons can Constrain Unassociated Elements

A M_1 level OCL constraint cannot consider unknown M_1 model elements - model elements are unknown if they do not exist yet when specifying the constraint. On the contrary, any unknown element becomes involved in a context-based constraint simply by having the matching context property value(s). Hence, the constrained M_1 elements can change without modifying the M_1 level CoCon expression. The indirect selection of constrained elements is particularly helpful in highly dynamic or complex systems. Every new, changed or removed system element is automatically constrained by a CoCon due to the element's context property values.

An OCL constraint expressed on M_1 level can only refer to those M_1 level elements that are directly associated with the constraint *via (possibly nested) associations*. On the contrary, the scope of a CoCon is not restricted. A CoCon can refer to elements that are not necessarily associated with each other or which even belong to different models. OCL constraints are associated with a model element. CoCons, however, may *not* necessarily be directly associated with a model element. Instead, one CoCon can *indirectly* select its constrained elements via the context property values associated with the model elements. A CoCon *can* directly be associated with model element, but it should not be associated with an constrained element that is indirectly selected via a context condition because the CoCon might not constrain the element anymore if the element's context property values change. By using CoCons, we don't have to understand every detail ('glass box view'). Instead, we only must understand the context property values we use for describing the elements involved in the requirement. The person who specifies requirements via CoCons does not have to have the complete knowledge of the system due to the *indirect* association of CoCons to the system parts involved. It can be unknown which elements are involved in the requirement when writing down the CoCon. The elements involved in the requirement can be identified automatically each time when checking the system for compliance with the CoCon.

All in all, we do not claim that CoCons are better than OCL. OCL is the best choice for expressing *normal* constraints in UML on a high detail level. CoCons are an additional concept if you need a more abstract solution (see section 4.1), if you want to check your model for compliance with the requirements without tests at runtime (see section 4.2), and if you need to keep track of a crosscutting requirement that relates to several, possibly unassociated model elements (see section 4.3).

> "You need to be more convincing as to the precision one can achieve with constraint specification using CoCons. OCL achieves a good level of precision with its constraints, making the results relatively deterministic. However, the kinds of capabilities you describe in terms of constraining unassociated elements bring to mind questions relating to determinism. "

As explained in the previous section 4.1, the UML-specific semantics of a CoCon-Predicate are defined in OCL at *metamodel* level. This means: we can express CoCons in OCL. Therefore, CoCons have the same determinism as OCL. The complexity of algorithms for detecting which system artefact elements violate which CoCon and of algorithms for detecting contradicting CoCons are discussed in [8].

5. INTEGRATING COCONS INTO UML

The notion of context-based constraints (CoCons) is not part of the UML at present. This section discusses how to integrate CoCons into the UML metamodel. Moreover, it explains why UML hardly can express how to weave in a constraint.

5.1 The Easy Part: Using UML's Constraints and Tagged Values

UML **profiles** provide a standard way of using UML in a particular area without having to extend or modify the UML metamodel. A profile is a predefined set of stereotypes, tagged values, constraints, and notation icons that collectively specialize and tailor UML for a specific domain or process. In order to understand this section, the reader should be familiar with the 'core' and 'extension mechanisms' packages of UML 1.5 (see [24]), or with the 'constraint' and 'profiles' packages of UML 2.0 (see [25]).

Context properties can be expressed in UML as tagged values. In UML 1.5, the metaclass 'TagDefinition' defines the name and other properties of a tagged value. A 'TaggedValue' belongs to exactly one TagDefinition and contains the actual values for a model element. A ≪ContextPropertyTag≫ TagDefinition specifies a context property name, e.g. 'Workflow', and a ≪ContextPropertyValue≫ TaggedValue (see figure 3) specifies a context property value, e.g. 'Create XYZ Report'. In UML 2.0, a Stereotype may have Properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties are referred to as tagged values.

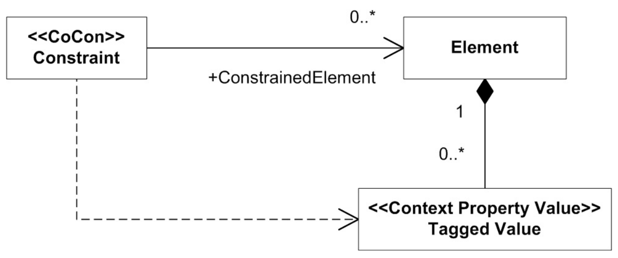

Figure 3: A Point Cut indirectly associates a Constraint with its Constrained Elements

Expressing context-based constraints in UML seems to be straightforward: In UML 1.5, a Constraint has the attribute 'body' which stores a BooleanExpression that must be true when evaluated for a model to be well-formed. In case of a ≪CoCon≫ Constraint (see figure 3), the body attribute stores strings that comply with the Context-Based Constraint Language (CCL) syntax definitions given in [7]. In UML 2.0, the CCL string is stored as ValueSpecification of the ≪CoCon≫ Constraint. But even though UML has a Constraint metaclass, it cannot easily integrate CoCons as explained in the next section.

5.2 The Problem: UML Constraints Don't Consider Point Cuts

The UML way of specifying *which elements are constrained by a constraint* doesn't fit to the notion of crosscutting constraints – the UML does not consider constraints which are *weaved in* as explained next. Both in UML 1.5 and in UML 2.0, a Constraint is associated with ModelElements via the 'constrainedElement' association. According to [25], this association defines the ordered set of Elements referenced by this Constraint. The constrained elements are those elements required to evaluate the constraint. An association *directly* links model element. Hence, the constrainedElement association holds a list of *directly* identified ModelElements affected by the Constraint. But, a ≪CoCon≫Constraint can be associated *indirectly* with model elements via a context condition. A CoCon constrains all elements fulfilling its context conditions even if these elements are not associated with each other at all.

In general, UML hardly can express a crosscutting advice as constraint because it can only express where to weave in the advice by directly listing the constrained elements. Point cuts, on the contrary, can define a condition, which is used to select the involved elements — the condition is a query, not a list of query results. UML cannot handle pointcut conditions whose join points (=constrained elements) are UML model elements because when specifying the constraint, we only know those join points which currently fit the point cut condition. We could list these known join points in the set of constrainedElements, but as soon as the system (model) changes, some of these join points may not fulfil the pointcut condition anymore, while some new join points may be added to the constrainedElements set. A UML tool which expresses crosscutting concerns as constraints must re-evaluate the constraint's pointcut condition after each model modification in order to keep the list of constrainedElements up to date.

In figure 3, the 'indirect association' of a CoCon with its constrained elements is depicted as dependency (a dotted arrow). It works as follows:

1. The context condition in the body expression of a ≪CoCon≫ Constraint refers to TaggedValue(s).
2. The TaggedValue is associated with a ModelElement. This ModelElement is *indirectly associated* with the CoCon if the following condition holds: this TaggedValues meets the CoCon's ContextCondition, while other TaggedValues associated with the same ModelElement must not violate that ContextCondition.

Maybe, some future version of UML supports 'indirect associations' whose pointcut condition must be re-evaluated each time when traversing them.

5.3 Proof of Concept Tool

The application of CoCons during modelling component-based system via UML has been evaluated in a case study being carried out in cooperation with the ISST Fraunhofer Institute, the Technical University Berlin and the insurance company Schwäbisch Hall. The goal of this case study was designing a complex component-based system for Schwäbisch Hall. During three years, the analysis of the requirements resulted in the 22 CoCon predicates and the Context-Based Constraint Language CCL. The 'CCL plugin' for the open source CASE tool ArgoUML has prototypically been implemented and is available for download at `ccl-plugin.berlios.de/`. It enables user to enter CCL expressions, and it integrates the verification of these CCL expressions into the Design Critiques ([32]) mechanism of ArgoUML. This mechanism continuously runs as a background process in ArgoUML. It iterates over all model elements and reports constraint violations or design recommendations. The CCL-Plugin adds design critiques that identify which model elements violate which CoCon for accessability CoCons and for distribution CoCons. These design critiques use the UML-specific CoCon semantics defined in section 4.1. Moreover, the CCL-Plugin adds design critiques that identify which CoCons contradict each other to ArgoUML. Hence, it demonstrates how to verify UML models for compliance with CoCons.

Two other proof-of-concept implementations enforce CoCons in enterprise Java Beans (EJB) systems at runtime. The EJBcomplex framework described in [23] uses dynamic proxies to intercept communication between EJB components. For instance, it can control which bean is allowed to invoke which other bean according to the current context of the caller and the callee. As brute force approach we could monitor simply every communication call in the system. In order to prevent overhead, we describe how to intercept only those communication calls via which are relevant for a certain CoCon in in [23]. Instead of using dynamic proxies as interception mechanism, we could also use aspect-oriented programming as examined in [30]. JBoss AOP and AspectWerkz support metadata in their current versions. The upcoming version of AspectJ will support metadata by modifying the AspectJ language.

> "Did the author ever apply CoCons to a real system? "

Yes, the pharmaceutical document management system dd-pro by the imphar AG uses CoCons to achieve multi-client capability. However, the imphar AG did not check their few UML models for compliance with their CoCons as discussed in this article. Therefore, this real life application neither proofs the claims of section 4 nor of section 5. Nevertheless, here is dd-pro's CoCon implementation in a nutshell: dd-pro controls documents. The user role of the current user *and the client where this current user is employed or for whom the current user is authorized to work* define the the access permissions for each document. As a result, the user john can only see and edit patent documents from client X, while user cathy only see and edit patent documents from the clients X, Y and Z (she is employed at X and authorized to work for Y and Z). This multi-client capability of dd-pro is achieved via CoCons. Context properties are managed on three levels:

database level: Different types (= classes) of documents exist. In dd-pro, the object-relational database schema is enhanced with context properties which tell the 'operational area' (see section 2.2) of each class. For instance, all classes belonging to the patent department have the operational area 'patents'. Moreover, dd-pro stores for each document to which client this document belongs.

user profile: The admins can configure the following context properties of each user:

- Employment: The client where user X is employed is called the employer of X. One user works at exactly one client, and one client can have many employees.
- Client-Authorization: If user X can work on behalf of client Y then X is called authorized user of client Y. A client on behalf of whom the user X is allowed to work is called the represented client of X. One user can work on behalf of many clients, and one client can have many users. If client Y employs user X that X is automatically an authorized user of Y.
- Role: A user can have the user roles. For instance, each Operational Area (see database schema level) an Editor role exists who can edit all records of the Operational Area from that client. Examples: Patent-Editor, Controlling-Editor, Human-Ressources-Editor.

runtime: At runtime, the user can select the value of his or her currentClient context property.

Depending on these context properties, dd-pro ensures that a user only can access certain documents according to the documents Operational Area, to the users Client-Authorization and Client-Role, and to the users Current Client.

> "We doubt whether the customer can understand (and is interested in) constraints on component level."

Both running examples in section 1.2 and 1.3 address only components in order to restrict this paper to a certain focus. When discussing requirements with Schwäbisch Hall, they understood and were interested in the component level. But as outlined above, the CoCons in the dd-pro system don't refer to components at all. Instead, they refer to documents and users.

Example for a crosscutting requirement which does not refer to components: `ALL DOCUMENTS WHERE Operational Area = Patent MUST BE WRITEABLE TO ALL USERS WHERE user.CurrentClient = document.owningClient AND user.Role = Patent-Editor`.

6. RELATED WORK ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT

6.1 Aspects at Source Code Level

Even though CoCons can be implemented via aspect-oriented frameworks, CoCons add the new notion of context-based point cuts as explained next. The places where to weave in an aspect are expressed in many different ways by current aspect-oriented languages. A few examples are the join point mechanism of AspectJ, the hyperspace mechanism, or the composition filtering mechanism. Modelling a pointcut is basically about modelling a selection query. The query defines a condition for selecting join points. The currently most common way to capture join points utilizes the implicit properties of program elements, including static properties such as method signature and lexical placement, as well as dynamic properties such as control flow. Typically, a pointcut query quantifies over properties of the source code (called 'internal context' in section 2.1).

On the contrary, a CoCon defines its pointcuts via context conditions. Such a context condition quantifies over context properties in order to select the join points. The context condition is a query that selects the join points where to weave in the crosscutting concern $C(x, y)$. As explained in section 2.1, the context doesn't have to be part of the source code or managed by the system. Likewise, [22] discusses that signature-based pointcuts cannot capture transaction management or authorization because there might be nothing inherent in an element's name or signature suggests transactionality or authorization characteristics.

"The example constraint shown in section 1.2could be implemented using the declare error feature in AspectJ. Why is it essential to use CoCons instead?"

Aspect-oriented software engineers have to define the relationships between aspects and their target artefacts. We suggest using context as glue between advices and joining points for two reasons. Contexts are implementation-independent and may express the **intention** why to weave in an advice better than normal pointcuts referring to source code properties. For instance, a business expert probably can tell whether an advice must affect all system element in the context 'sales department' or all elements in the context of the 'purchase a ticket' workflow , but this expert will hardly know which regular expression a methods or components should match in order to be affected. Moreover, we can identify and refer to contexts even before the first line of source code has been written down. For instance, stakeholders can understand and negotiate the privacy policy of section 1.2 without knowing which components actually exist already or will exist. As soon as some source code or binary is added to the system that matches a CoCon's context condition, it will be affected by the CoCon.

In [22], several mechanisms are examined for referring to metadata in pointcuts. In [33], aspects are expressed as C# custom attributes. The aspects are weaved in using introspection and reflection technique *based on metadata* in the .NET common language runtime. Hence, recent research examines context-based aspects. CoCons add two suggestions: we can express our context-based aspect in an abstract textual language that does not refer to the source code level at all. Furthermore, we can manage the metadata outside of the system/source code in an external repository. We used external repositories in [23, 30] because we wanted our frameworks to work without modifying the components.

6.2 Aspects at Design Level

With regards to considering aspects already during design, several interesting approaches exist. In [2], crosscutting concerns are also expressed at a high abstraction level during design. But, in this approach the pointcuts are defined by listing the involved UML models. Instead, a CoCon indirectly selects its constrained elements according to their context properties.

A graphical way to model join points called 'Join Point Designation Diagram'(JPDD) is introduced in [35]. JPDDs describe 'selection patterns' which specify all properties a model element (i.e., UML Classifier or UML Message) must provide in order to represent a join point. The semantic of JPDDs is specified by means of OCL Expressions. JPDD could be used to model CoCons if the context properties are expressed as tagged values for each model element. But still, the JPDD approach demands to change the UML metamodel each time when a pointcut condition is changed or added. Furthermore, the JPDD community doesn't consider context in pointcuts yet.

In the hyperspace approach discussed in [37], a hyperslice encapsulates a crosscutting concern. The hyperslices are composed to form a complete system: two hyperslices can be composed by a hypermodule, which contains correspondence rules that determine at what points the hyperslices should be joined. This approach is reflected in the Hyper/J framework. A CoCon expresses a hyperslice. But, CoCons have a different notion of composition rules: In Hyper/J, the composition rule indicates which elements in the hyperslices describe the same concepts, and how these elements must be integrated. The elements describing the same concept are selected via a query. For instance, the composition rule could select 'all elements having the same name'. An aspects typically selects its join points according to their structural properties, like their name. On the contrary, a CoCons selects its constrained elements (=join points) according to their context properties.

An similar approach for composing (= weaving) hyperslices (=advices) into UML models is presented in [10]: so called *composition relationships* identify overlapping elements in different UML models and specify how to integrate these elements. Again, the composition relationships discussed up to now don't select the involved elements according to the element's context. But, they could because they are expressed in OCL, which can refer to tagged values, which can express the element's context.

In a distributed system, objects that interact heavily should be located together. The aspect-oriented approach D^2AL groups collaborating objects that are directly linked via associations. It describes in textual language in which manner these objects interact which are connected via these associations. This does not work for objects that are not directly linked like 'all objects needed in the 'Create Report' workflow. Darwin (or 'δarwin') is a *configuration language* for distributed systems described in [28] that, likewise, expresses the architecture explicitly by specifying the associations between objects. However, there may a reason for allocating objects together even if they do not collaborate at all. For instance, it may be necessary to cluster all objects needed in a certain workflow regardless whether they invoke each other or not. Distribution CoCons introduced in [5] allocate objects together because of shared context instead of direct collaboration.

6.3 Early Aspects at Requirements Level

Aspects already exist in requirements and architecture artefacts. According to [1], an early aspect is a crosscutting requirement because it recurs in several stakeholders' or viewpoints' requirement specifications. But, even if one requirement is mentioned in multiple requirement specifications, it won't necessarily be implemented in different classes. Furthermore, a requirement, which is only mentioned once in the requirement specification documents, can still become a crosscutting concern when implementing the system. CoCons differ from the common notion of early aspects as discussed in [6], because a CoCons does not need to be mentioned in several places in order to express a crosscutting requirement. Instead, it expresses an aspect at one place. Its context conditions help to trace this aspect to the other system artefacts at design, source code or runtime level because the context conditions will select the constrained elements in each artefact.

According to [4], we should avoid the word 'all' when stating a requirement because it is an example of a hard to interpret requirement. This may be right, but its also is an interesting requirement because it may become a crosscutting requirement. A CoCon use the word 'all' to express that there may be more than one join point for this requirement, and it describes its involved system elements (= join points) indirectly via their context. By using dangerous 'all' statements, we can write down the crosscutting concern at one place in the requirement specification and, thus, avoid redundancy, which may impede us in evolving our system.

"The abstract implies that an RE aspect is a single requirement. Do you not consider crosscutting concerns at requirements level which are described by multiple requirements?"

No. CoCons don't handle if someone expresses the same requirement several times in different requirement specifications. This question is addressed in the Theme approach described in [11] which consists of two parts: Theme/Doc identifies so called 'early aspects' via linguistic analysis of plain English requirements specifications: a requirement which is mentioned several times in the requirements specification is called 'early aspect' because its somehow crosscutting the requirements specification. But this horizontal notion of early aspects doesn't care whether this 'crosscutting-at-requirements-level' requirement is implemented in one single place or not - this notion ignores if the implementation of the requirement will be crosscutting in the AspectJ understanding of 'crosscutting'. On the contrary, CoCons express vertical early aspects: they express features which are mentioned only once in the requirements specification but affect possibly many places in the model or in the source code. The Theme approach also has a second part called Theme/UML which examines vertical aspects (= aspects which are crosscutting at the source code level). It expresses these early aspects via UML. For each aspect, a list of all its join points is compiled. On the contrary, a CoCon doesn't list each constrained element. Instead, a CoCon indirectly describes its join points via the join point's context properties.

The PROBE framework described in [20] defines which aspect crosscuts which requirement by writing down composition rules. Again, these rules list all affected requirements. On the contrary, a CoCon doesn't list each constrained element.

7. CONCLUSION

7.1 Limitations of CoCons

Taking only the metadata of an element into account bears some risks. It must be ensured that the context property values are always up-to date. If the metadata is extracted newly each time when checked and if the (automatical) extraction mechanism works correctly then the metadata is correct and up-to-date. Moreover, the extraction mechanism ensures that metadata is available at all. If the metadata cannot be extracted automatically, we recommend that the quality assurance department approves the manually encoded metadata and defines an expiration date after which the metadata must be approved newly.

"The author mentions that keeping the context property values up to date is an issue, but I think it is a big issue."

Yes. That's why a whole chapter deals with maintaining context property values in [7].

"The motivation of the work as described in section 1 is not convincing: In section 1.4, the author states that one of the goals of the approach is to allow stakeholders who do not know anything about the code to specify join-points. However, if CCL is to be used in source code level, one still needs to go through the source code and add the context information. "

Yes, but those people who add the context information are different to those who write down the CoCons. The experts who understand a system artefact enrich this artefact with context properties and enable the requirements engineers to hide the technical details.

Within one system, only one context property terminology should be used. For instance, the workflow 'New Customer' should have exactly this name (and semantics) in every part of the system, even if different companies manufacture or use its parts. Otherwise, string matching gets complex when checking a context condition.

7.2 Benefits of CoCons

CoCons select their constrained system elements via the element's context properties. In contrast to other grouping techniques, e.g. packages or stereotypes, context properties can *dynamically* group elements even at runtime. Furthermore, the assist in handling sets of elements that share a context even across different element types, artefact types, or platforms. They also help to express crosscutting requirements relating to several elements that are not associated with each other or even belong to different artefacts.

The same CoCon used to check the system model can also be used to check other system artefact for violated or contradicting requirements because CoCons specify requirements at an artefact-type-independent, abstract level. Therefore, they enable us to check different software development artefacts for compliance with the same CoCon during modelling, during deployment and at runtime. This paper has suggested how to check UML models for compliance with CoCons.

The requirements specification should serve as a document understood by designers, programmers, and customers. CoCons can be specified in easily comprehensible, straightforward language that assists every English speaking person in understanding their design rationale. A CoCon can be translated into an artefact-specific advice that is more complex than the corresponding CoCon because it refers to all the artefact-specific details. The effort of writing down a requirement in the minutest details is unsuitable if the details are not important. CoCons facilitate staying on an abstract level that eases requirements specification.

The people who need a requirement to be enforced do often neither know all the details of every part of the system (glass box view) nor do they have access to the complete source code, model or configuration files. It can be unknown which elements are involved in the requirement when we specify it via CoCons. Software tools that check the system artefacts for violated or contradicting CoCons will identify those elements that are involved in the requirement automatically. CoCons help us to specify requirements because it is easier to write down a requirement if we don't have to list all of the elements that relate to this requirement. By using CoCons, we don't have to understand every detail of the system. Instead, we only need to understand the context property values we use for describing the context of the system elements.

When adapting a system to new requirements, existing dependencies and invariants should not be violated. CoCons help us to ensure consistency during system evolution. A context-based constraint serves as an invariant and, thus, prevents the violation of requirements during modifications of the system artefacts. It assists in detecting when design or context modifications compromise intended functionality. Hence, CoCons help us to prevent unanticipated side effects during (re-)design, during (re-)configuration and at runtime. Requirements tend to change quite often. Indirectly selecting the elements involved improves adaptability because every new or changed element is constrained automatically if it fits to the context condition. The context property values can be easily adapted whenever the context of an element changes. Furthermore, each modified or additional CoCon can automatically be enforced and any resulting conflicts can be identified. 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 crosscutting requirements.

8. REFERENCES

[1] J. Araújo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdoğan. Early aspects: The current landscape. Technical Report COMP-001-2005, Lancaster University, February 2005.

[2] J. Araújo, A. Moreira, I. Brito, and A. Rashid. Aspect-oriented requirements with UML. In M. Kandé, O. Aldawud, G. Booch, and B. Harrison, editors, *Workshop on Aspect-Oriented Modeling with UML*, 2002.

[3] AspectJ. http://www.aspectj.org.

[4] D. M. Berry and E. Kamsties. The syntactically dangerous all and plural in specifications. *IEEE Software*, 22(1):55–57, January 2005.

[5] F. Bübl. What must (not) be available where? In R. Meersman, Z. Tari, and D. C. Schmidt, editors, 5^{th} *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[6] F. Bübl. Never mind the source code, but be aware of the context when dealing with cross-cutting requirements. In *Early Aspects Workshop*, October 2005.

[7] F. Bübl. Tracing crosscutting requirements for component-based systems via context-based constraints. PhD Thesis, Technical University Berlin, Germany, 2005.

[8] F. Bübl and M. Balser. Tracing cross-cutting requirements via context-based constraints. In H. Yang, editor, 9^{th} *Conference on Software Maintenance and Reengineering, Manchester, Great Britain*. IEEE computer, March 2005.

[9] J. Cheesman and J. Daniels. *UML Components*. Addison-Wesley, 2000.

[10] S. Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1):71–100, 2002.

[11] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design - The Theme Approach*. Addison-Wesley, Reading, 2005.

[12] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, , and A. Wills. The amsterdam manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, 1999.

[13] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, and A. 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.

[14] K. Devlin. *Logic and Information*. Cambridge University Press, New York, 1991.

[15] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.

[16] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In R. Arnold and S. Bohner, editors, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[17] G. Hirst. Context as a spurious concept. In *Proceedings of the third workshop on Conference on Intelligent Processing and Computational Linguistics, Rio de Janeiro, Mexico City*, pages 273–287, 2000.

[18] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.

[19] V. Kashyap and A. P. Sheth. Schematic and semantic similarities between database objects: A context-based approach. *Very Large Data Bases (VLDB) Journal*, 5(4):276–304, 1996.

[20] S. Katz and A. Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *International Conference on Requirements Engineering (RE), Kayoto, Japan*, pages 48–57. IEEE Computer Society, 2004.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming ECOOP*, volume 1241 of *LNCS*, pages 220–242, Berlin, 1997. Springer.

[22] R. Laddad. AOP at work: AOP and metadata: A perfect match, part 1. Technical report, DeveloperWorks, 2005.

[23] A. Leicher, A. Bilke, F. Bübl, and E. U. Kriegel. Integrating container services with pluggable system extensions. In R. Meersman, Z. Tari, and D. C. Schmidt, editors, 5^{th} *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[24] OMG. UML 1.5, formal/03-03-01, March 2003.

[25] OMG. UML 2.0 infrastructure specification, ptc/03-09-15, September 2003.

[26] J. D. Palmer. Traceability. In R. H. Thayer and M. Dorfman, editors, *Software Requirements Engineering*, pages 412–422. IEEE Computer Society, 2000.

[27] F. A. C. Pinheiro. Formal and informal aspects of requirements tracing. In *Proceedings of the third workshop on Requirements Engineering, Rio de Janeiro, Brazil*, 2000.

[28] M. Radestock and S. Eisenbach. Semantics of a higher-order coordination language. In *Coordination 96*, 1996.

[29] B. Ramesh and M. Jarke. Toward reference models of requirements traceability. *Transactions on Software Engineering*, 27(1):58–93, 2001.

[30] F. Ratzlow. Einsatzmoeglichkeiten der Aspekt-orientierten Programmierung im Kontext der Java 2 Enterprise Architektur. Diploma Thesis, FHTW Berlin, Germany, Prof. Ingo Classen, 2004.

[31] M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In A. Evans and S. Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*. Springer, Berlin, LNCS, 2000.

[32] J. E. Robbins and D. 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.

[33] W. Schult and A. Polze. Aspect-oriented programming with c# and .net. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 241–248, 2002.

[34] A. P. Sheth and S. K. Gala. Attribute relationships: An impediment in automating schema integration. In *Proc. of the Workshop on Heterogeneous Database Systems (Chicago, Ill., USA)*, December 1989.

[35] D. Stein, S. Hanenberg, and R. Unland. Modeling pointcuts. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.

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

[37] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. Degrees of separation: Multi-dimensional separation of concerns. In *Int. Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[38] J. B. Warmer and A. G. Kleppe. *Object Constraint Language – Precise modeling with UML*. Addison-Wesley, Reading, 1999.