Never mind the Source Code, but be aware of the Context when dealing with Cross-Cutting Requirements

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

Abstract

One crosscutting requirement (also called aspect) affects several parts of a software system. It is difficult to express aspects during requirements analysis or at design level if we don't know all implementation details yet. For instance, it is difficult to determine at which places (= join points) which aspect must be added to (= weaved in) the system. In this paper, we suggest to express aspects in a way that is understandable for stakeholders and customers. We achieve this by using a new notion of weaving: We determine where to weave in which aspect by considering the system's context – our pointcuts are context-based. We express aspects via Context-Based Constraints (CoCons) independent of programming or modelling languages.

1. Introduction

Before going into details, we start with a crosscutting requirement example.

1.1. Privacy Policy Example

Let's assume that the component-based software system 'pet shop' 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' because a XYZ report must not contain personal data.

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?

1.2. Goal: Identify Join Points without knowing the Source Code

According to [12], **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.

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 without knowing the source code at all. For example, we want to determine where to control the privacy policy (see section 1.1) without knowing any technical detail about the system's components.

1.3. Our Context-Based Approach in Brief

We specify crosscutting requirements via context-based constraints (CoCons). They are formally defined in [7]. This paper compares them with aspects. The basic notion of CoCons can be explained in just a few sentences:

1. We annotate the system artefact elements with formatted metadata called 'context properties'. 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 via their context properties – its pointcuts 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 1.1 can be expressed via the CoCon *all elements that handle personal data must be inaccessible to those elements that are used in the Create XYZ Report workflow*.

2. Introducing Context Properties

2.1. What is Context?

Each element resides in an infinite number of contexts – according to [18, 10], it is impossible to list all contexts of an element because we can't completely define what an element denotes. All context definitions developed in computer science fail to provide a *general* theory of context as discussed in [9]. Only limited context models can be handled. Thus, we stick to a simple and limited notion:

- Context that is not part of or managed by the system can be taken into account.
- The context of a context is ignored here.
- The context of a element characterizes the situation(s) in which the element resides as defined in [8].

2.2. Context Properties: Formatted Metadata Describing Elements

We suggest expressing context as metadata using 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 workflows in which the associated element is used.
- The values of the context property 'Personal Data' signal whether an element handles data of private nature.
- The values of the context property 'Operational Area' describe, in which department(s), module(s), or domain(s) the associated element is used. They provide an organisational perspective.

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. Instead, 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.

3. Context-Based Constraints (CoCons) express Cross-Cutting Requirements

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. In terms of AspectJ ([3]), $C(x, y)$ expresses an advice. 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 1.1). Another CoCon-predicate can express that certain elements must (or must not) be allocated to certain computers (distribution requirement). All in all, 22 CoCon-predicates for component-based systems are defined in [6]. 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. A CoCon could relate many *sets* of constrained elements. We only discuss CoCons that relate *two* sets of elements 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. A **context condition** selects artefact elements according to their context property values. Different query languages exist for expressing a context condition - the right choice depends on in which format the 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 [6]. 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 1.1 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. We use a two-step approach for defining semantics of a CoCon-predicate:

- The **artefact-type-independent semantics** of a CoCon-predicate do not refer to specific properties of an individual artefact type. It simply expresses the semantics in plain English. For instance, 'x must be accessible to y' is an artefact-type-independent semantics for $C(x, y)$.
- 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 UML-specific semantics of `ACCESSIBLE TO` CoCons consider a lot more details for each of the many diagram types of UML.

3.3. Proof of Concept Tools

We have build proof-of concept tools for different artefact types. 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. As a result, 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 integrates the verification of CoCons into the Design Critiques ([16]) mechanism of ArgoUML. It adds design critiques that identify which model elements violate which CoCon, and it adds design critiques that identify which CoCons contradict each other. Hence, it demonstrates how to verify UML models for compliance with CoCons.

Two other prototypes enforce CoCons in enterprise Java Beans (EJB) systems at runtime. The EJBcomplex framework described in [14] 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. Instead of dynamic proxies as interception mechanism, we could also use aspect-oriented programming as examined in [15]. JBoss AOP and AspectWerkz support metadata in their current versions. The upcoming version of AspectJ will support metadata by modifying the AspectJ language. Furthermore, we could also use the pet shop's filter mechanism. Different artefact-specific semantics for Java applications exist. But, the stakeholders who have to understand and agree to the crosscutting requirements typically don't care for artefact-specific implementation details. Using CoCons, they only have to understand the abstract, artefact-independent semantics and the system's context.

3.4. Comparing CoCons with 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. Modeling a pointcut is basically about modeling 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.

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 cross-cutting 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, [13] 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.

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 'purchase 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 con-

texts even before the first line of source code has been written down. For instance, stakeholders can understand and negotiate the privacy policy of section 1.1 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 [13], several mechanisms are examined for referring to metadata in pointcuts. In [17], 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 [14, 15] because we wanted our frameworks to work without modifying the components.

3.5. Comparing CoCons with 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 [19]. 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.

3.6. Comparing CoCons with 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, because a CoCons does not need to be mentioned in several places in order to express a cross-cutting 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 [5], 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 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 Theme approach described in [4] identifies early aspects via linguistic analysis of requirement documents and expresses these early aspects via UML. For each aspect, a list of all join points is compiled. On the contrary, CoCons don't list their constrained elements. Instead, CoCons indirectly describe their join points via their context.

The PROBE framework described in [11] 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.

4. Conclusion

4.1. 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 extraction mechanism works correctly then the metadata is correct and up-to-date. Moreover, the extraction mechanism ensures that metadata is available at all.

Within one system, only one terminology for context property values should be used. For instance, the workflow 'Add 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.

4.2. Benefits of CoCons

Context properties assist in handling sets of elements that share a context even across different element types, artefact types, or platforms. They also help to express 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 validate different software development artefacts for compliance with the same CoCon during modelling, during deployment and at runtime.

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.

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] E. Baniassad and S. Clarke. Finding aspects in requirements with theme/doc. In *Early Aspects Workshop*, 2004.

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

[6] F. Bübl. The context-based constraint language CCL for components. Technical Report 2002-20, Technical University Berlin, Germany, October 2002.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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