Tracing Cross-Cutting Requirements via Context-Based Constraints

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

Michael Balser
Universität Augsburg, Germany
Michael.Balser@informatik.uni-augsburg.de

Abstract

In complex systems, it is difficult to identify which system element is involved in which requirement. In this article, we present a new approach for expressing and validating a requirement even if we don't precisely know which system elements are involved: a context-based constraint (CoCon) can identify the involved elements according to their context. CoCons support checking the system for compliance with requirements during (re-)design, during (re-)configuration or at runtime because they specify requirements on an abstract level independent of the monitored artefact type. They facilitate handling cross-cutting requirements for possibly large, overlapping or dynamically changing sets of system elements - even across different artefact types or platforms. Besides defining CoCons, we discuss algorithms for detecting violated or contradicting CoCons.

1. Introduction

1.1. Continuous Requirements Tracing

The context for which a software system was designed changes continuously throughout its lifetime. When adapting a system to new, altered or deleted requirements, existing requirements should not unintentionally be violated. However, no single architect has all the knowledge needed for checking every new, removed or changed system element for compliance with every existing and relevant requirement.

Gotel defines requirements traceability in [9] as 'the ability to describe and follow the life of a requirement in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)'. Providing traceability in requirements documentation facilitates managing change because it enables us to consider which system element is involved in which requirements.

1.2. Privacy Policy Example

As an example for a crosscutting security requirement, let's assume that our 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' because a XYZ report must not contain personal data.

Which elements of which system artefact should be checked for whether they do (or do not) access which other elements? For instance, which model element(s) of which UML diagram should be checked for what details?

1.3. Availability Requirement Example

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

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

Due to this requirement, any computer of the controlling 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, next sections outline the problems we want to solve.

1.4. Goal: Independent of Artefact Types

In a system model, one requirement can affect several model elements that may not be associated with each other or may even belong to different diagram types. In source code files, one requirement can be reflected in many different lines of code. Likewise, one requirement can affect several places in configuration files. At runtime, one requirement can affect 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. Instead, we try to express 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. 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.

1.5. Goal: Adaptive Approach

One requirement can affect many system elements. Writing down one requirement directly for each individual element involved is expensive if the involved elements change frequently or if many elements are involved in the requirement. Firstly, it is expensive to check all elements involved whether to adapt them each time the requirement changes. Secondly, it is expensive to check the system if the changed requirement newly applies to any other elements that were previously not involved in it. The system gets less comprehensible if the same requirement is specified at every affected place. Redundancy causes information overload and can result in inconsistency.

Hence, we try to address *all* of the involved elements in *one* requirement expression, and we try to describe the involved elements in an **adaptive** way that allows for automatic identification of all the involved system elements. Large-scale or frequently changing systems can be more easily checked for compliance with adaptive requirements specification because the elements involved in a requirement can automatically be determined. For example, the adaptive specification of the example privacy policy should enable us to identify those components that handle personal data and those components used in the workflow 'Create XYZ Report' automatically.

1.6. Goal: Detect Violated or Contradicting Requirements

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

In order to enable tools to understand our requirements, we specify them via constraints.

1.7. The New Constraint Technique 'CoCons' in Brief

Requirements specification via context-based constraints (CoCons) have been informally introduced in [2]. This paper presents a heavily revised notion of CoCons. Furthermore, it outlines their formal semantics and discusses how to detect violated or contradicting CoCons. The basic notion of CoCons can be explained in just a few sentences:

1. Metadata is data about data . 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 is a constraint that expresses a requirement. It can select its constrained elements via their context properties.
3. A CoCon expresses a condition on how system elements must (or must not) relate to each other. We only consider CoCons that relate two sets of elements with each other.

For instance, we could use context properties to mark each element that handles personal data, and to mark each element that is is used in the 'Create XYZ Report' workflow. Then, the privacy policy in section 1.2 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 inifinite number of contexts – according to [18, 11], 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* theorie of context as discussed in [10]. Only limited context models can be handled. Thus, we stick to a simple and limited context model:

- 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].

This notion needs a precise definition of 'situation'. In situation calculus ([7]), **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. Instead, it only characterizes the situation.

2.2. Context Properties: Formatted Metadata Describing Elements

The context of an 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 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.

2.3. Research Related To Context Properties

A concept similar to context properties was discussed in the 90ties: database objects are annotated via intensional description called 'semantic values' ([18]) in order to identify those objects in different databases that are semantically related. Likewise, context properties are annotated to elements in order to determine the relevant element(s). However, the semantic value approach has a different purpose and, thus, a different notion of *relevant*: the purpose of semantic values is to identify semantically related objects in order to resolve schema heterogenity among them.

Likewise, 'domains' ([19]) are similar to context properties. Domains are typically used in policy management. In contrast to a context properties, a domain consists of a single name and not of a name and value(s). Moreover, domains are hierarchical.

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 *cross-cutting* 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 defines a new constraint technique called 'CoCons' for requirements specification.

3.1. Intuitive Definition of Context-Based Constraints

A **context-based constraint** (CoCon) is a constraint. It expresses a condition on how its constrained elements must relate to each other. This condition is called **CoCon-predicate**. Different CoCon-predicates exist. For example, a CoCon-predicate can express that certain elements must (or must not) be accessible to other elements (security requirement, see 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). All in all, 22 CoCon-predicates for component-based systems are defined in [1]. Future research hopefully will identify additional CoCon-predicates.

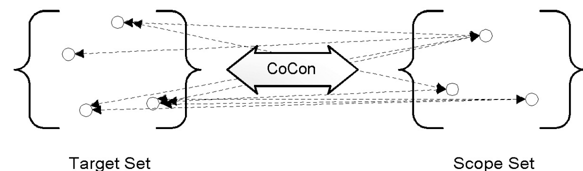

Figure 1. A CoCon Relates any Element of the 'Target Set' with any Element of the 'Scope Set'

One requirement can affect several possibly unassociated elements. A CoCon can indirectly select its constrained elements via context conditions. A **context condition** selects artefact elements according to their context property values. A CoCon can relate many *sets* of constrained elements. We only discuss CoCons that relate *two* sets of elements. Figure 1 illustrates that a CoCon relates each element of one set to each element of the other set and expresses a CoCon-predicate (depicted as dotted arrows) for each *pair* of related elements. The two sets related by a CoCon are called 'target set' and 'scope set'. The elements in these sets can be selected via context conditions.

The Context-Based Constraint Language CCL has been defined in [1]. 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 Report'
```

3.2. Two-Step Approach for Defining CoCon-predicate Semantics

This section discusses *how to* formally define the CoCon semantics. CoCons can be applied to artefacts at different development levels, like models at design level or component instances at runtime. Different artefact types can be used at each development level. 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. For instance, the artefact-type-independent semantics of an `ACCESSIBLE TO` CoCon are defined 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. We do not list these OCL expressions here because they consider a lot more details for each of the many diagram types of UML - they are about as long as this article. E.g., one OCL expression defines that a UML component diagram can express the invocation of a component via a dependency (a dotted arrow). If such a dependency exists between the components x and y in a certain UML component diagram then the x is `ACCESSIBLE TO` y in this artefact.

3.3. Formalization of Context-Based Constraints

A CoCon selects two sets of elements. Each element of its target set must relate to each element of its scope set as expressed via the following predicate logic formula:

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

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 ALLOCATED TO y` (designing distributed systems with CoCons is discussed in [3]). $C(x, y)$ is a binary relation – it expresses how x relates to y. In other mathematical notations, $C(x, y)$ can also be expressed as $(x, y) \in C$, or xCy. It would be possible to discuss n-ary CoCon-predicates. However, we focus on binary relations for reasons explained in section 4.2.

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. $TCC(x)$ represents the target set context condition: $TCC(x)$ defines a condition on the context property values of x. Likewise, $SCC(y)$ represents the scope set context condition: $SCC(y)$ defines a condition on the context property values of y.

The predicates $TR(x)$ and $SR(y)$ define the range of the relation. $TR(x)$ defines which types of elements can be contained in the target set, and $SR(y)$ defines which types of elements can be contained in the scope set. The range-predicates $TR(x)$ or $SR(y)$ can be omitted if they are the same. For example, only artefact elements of the type 'component' are discussed throughout this paper. In that case, we can leave out the predicates $TR(x)$ = 'x is a component' and $SR(y)$ = 'x is a component' because we only present formulas whose universe consists of the components.

A CoCon can have two operations on its CoCon-predicate $C(x, y)$:

- The operation `NOT` negates $C(x, y)$ as follows: $\forall x, y : TR(x) \wedge TCC(x) \wedge SR(y) \wedge SCC(y) \rightarrow \neg C(x, y)$
- The operation `ONLY` is mapped to two propositions:
  - $\forall x, y : TR(x) \wedge TCC(x) \wedge SR(y) \wedge \neg SCC(y) \rightarrow \neg C(x, y)$
  - $\forall x, y : TR(x) \wedge TCC(x) \wedge SR(y) \wedge SCC(y) \rightarrow C(x, y)$

The privacy policy example is mapped to predicate logic as follows:

- $C(x, y)$ is defined as 'x is accessible to y'.
- $TR(x) = SR(y)$ is defined as 'x is a component'
- $TCC(x)$ is defined as 'x handles personal data'
- $SCC(y)$ is defined as 'y is used in the 'Create Report'workflow'.

3.4. Which Requirements can be Expressed via CoCons?

CoCons can both express functional requirements like 'x must be notified of y' and non-functional requirements like 'x must be accessible to y'.

Is it possible to write down all requirements via CoCons? The simple answer is: No. A CoCon selects its constrained artefact element *types* via its range-predicates $TR(x)$ and $SR(y)$. For instance, a CoCon can select components. The expressiveness of a CoCon-predicate depends on the selected element type. 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'. But, not all system components have the same interface. Therefore, a CoCon restricted to a element type cannot express conditions on specific properties of one instance of this element type. Instead, a CoCon only can express conditions on the type properties.

Many other constraint languages exist for expressing conditions on properties of single elements. On the contrary, CoCon-predicates express conditions on how *two* constrained elements *relate to* each other.

The key new concept of CoCons is the indirect selection of the constrained elements according to their context properties. For example, the direct selection '$component_1$, $component_4$ and $component_7$' can describe the same elements as "*All components whose context property 'Personal Data ' has the value 'True'*". However, the indirect selection *automatically* adapts to changed elements or context changes, while the direct selection doesn't. For instance, eventually a new component will become part of the system after writing down the CoCon. The new $component_{31}$ is not selected by the direct selection given above. On the contrary, the indirect selection will automatically apply to $component_{31}$ as soon as $component_{31}$'s context property 'Personal Data' has the value 'True'. The indirect selection expression must not be adapted if system elements or their contexts change. Instead, the indirectly selected elements are identified by evaluating the context condition each time when the system is checked for whether it complies with the CoCon.

A CoCon can refer to elements that are unknown when specifying the CoCon because the CoCon will apply to a new or modified element as soon as the context property values of this element fit the CoCon's context condition. Therefore, CoCons can only express requirements for 'unknown elements' – all we know when writing down the CoCon is the type of the constrained element (e.g. component), its context and its demanded relationship to another 'unknown element'. As a result of this restriction, CoCons are *adaptive*: they can adapt automatically to changed systems or to changed contexts because the constrained elements are identified newly each time the compliance of the system with the CoCon is checked.

3.5. Research Related to Context-Based Constraints

According to [20], goals denote the objectives a system must meet. This paper describes an adaptive constraint language for expressing goals for possibly large groups of system elements and tracing them through all artefacts used in a software development process. When eliciting goals, the requirements engineers focus on the problem domain and the needs of the stakeholders, rather than on possible solutions to those problems. The person who specifies a requirement via CoCons does not have to have the complete (glass box view) knowledge of the system due to the *indirect* association of CoCons to the system parts involved. It can be unknown which element are involved in the goal when writing it down as a CoCon. Hence, CoCons facilitate tracing a high level goal to those system elements where the goal is realized.

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. According to [15], an **requirements allocation table** is the common mechanism used to maintain this information. However, keeping track of each individual requirement or element in such a table 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 a requirements allocation table that directly links requirements to individual elements. Instead, CoCons facilitate specifying requirements for possibly large groups of elements. They allow for indirect, *adaptive* selection of all the elements involved in the requirement.

Recent work by both researchers ([5]) and practitioners ([17]) has investigated how to model non-functional requirements and to express them in a way that is measurable or testable. Non-functional requirements (also known as quality requirements) are generally more difficult to express in a measurable way, making them more difficult to analyse. They are also known as the 'ilities' and have defied a clear characterisation for decades. In particular, they tend to be properties of a system as a whole, and hence cannot be verified for individual system elements. Moreover, they often are cross-cutting:a **cross-cutting** requirement violates the separation-of-concerns paradigm – it is not possible to handle this requirement at only one single, encapsulated place in the sytem. Via the two-step approach for defining the semantics, CoCons can express 'ilities' clearly. They are particulary helpful in expressing cross-cutting ilities because one CoCon can constrain several involved elements according to their context property values.

The first order logic based rule language CLIX is now

compared with CoCons. The xlinkit framework as presented in [13] monitors XML artefacts for compliance with CLIX expressions. CoCons can select their constrained elements via context conditions. Likewise, a CLIX rule can indirectly select the constrained elements via XPath queries. However, these XPath queries always refer to the artefact elements, while context conditions don't necessarily refer to the artefact elements. Instead, context conditions refer to the *contexts* of the artefact elements. It is possible to express CoCons via CLIX if the context properties of the artefact elements are stored in the monitored artefact. As explained in section 3.3, the context properties of an artefact element can be stored as 'external' metainformation in a different artefact than their element. If the attribute values of an element are not expressed in the checked artefact then XPath cannot refer to it. Context conditions cannot be expressed in XPath if they refer to context properties that are stored in another artefact.

A CoCon relates two sets of elements and defines a condition on each related pair of elements. Likewise, CLIX rules can select two sets of elements via XPath and define a condition on each related pair of elements. Two major differences between CoCons and CLIX exist, though. First, a CLIX rule always defines action semantics: it describes what to do if two elements violate or meet the CLIX rule. On the contrary, CoCons only define constraints and ignore actions and events. The major difference is the two-step semantics definition of CoCons: a CLIX rule applies to artefacts of one type, while a CoCon can apply to artefacts of many types. Thus, a CLIX rule can express the artefact-specific semantics of a CoCon, but there is no additional abstraction layer in CLIX/xlinkit that allows to express requirements for different artefact types.

Another recent approach can indirecly select the constrained targets: the Ponder language for specifying management and security policies is defined in [6]. Different families of policies can be expressed in Ponder. One of them are *authorisation policies*. They define what activities a member of the subject domain can perform on the set of objects in the target domain. For example, the privacy policy of section 1.2 can be defined in Ponder as:

```
  inst auth- privacyPolicy {
subject /componentsUsedInTheWorkflowCreateReport;
action access() ;
target /componentsHandlingPersonalData;}
```

Ponder differs from CoCons in several ways. Firstly, CoCons neither consider events nor actions, while Ponder has operational semantics. Secondly, Ponder uses domains (see section 2.3) in order to select the object to which a policies applies, while CoCons identify their constrained elements according to their context properties. Finally, Ponder has no two-step semantics definition. Similar to CLIX, Ponder misses this additional abstraction layer. CoCons facilitate handling *cross-cutting* requirements because one CoCon can constrain many elements of many system artefacts.

The next two sections discuss how to apply CoCons after they have been written down.

4. Detect CoCon Violations

4.1. CoCon-Violation Conflicts

A **CoCon-violation conflict** occurs if the relation between the artefact elements x and y does not comply with the CoCon-predicate $C(x, y)$. The next section presents an algorithm that enables software tools to monitor the system artefacts for CoCon violations automatically.

4.2. The Detect-CoCon-Violations Algorithm

This section presents a general algorithm for detecting CoCon-violation conflicts.

input : The finite set A containing all n elements $a_1, ..., a_n$ of the checked artefact and the CoCon $\forall x, y \in A : TCC(x) \wedge SCC(y) \rightarrow C(x, y)$

output : The binary result relation $R : A \times A$ containing those of pairs of artefact elements that violate the CoCon

`/* identify constrained elements */`
foreach $a_i \in A$ **do**
  if a_i *fulfils the target set context condition* $TCC(a_i)$ **then**
    add a_i to the scope set $TARGET$
  if a_i *fulfils the scope set context condition* $SCC(a_i)$ **then**
    add a_i to the targe set $SCOPE$

`/* check constrained elements */`
foreach $a_s \in SCOPE$ **do**
  foreach $a_t \in TARGET$ **do**
    if $C(a_s, a_t)$ *is violated according to the artefact-type-specific* $Semantics^{C(x,y)}_{Artefact-Type}$ **then**
      add the pair (a_s, a_t) to the result relation R

Algorithm 1: Detect-CoCon-Violations

As input, algorithm 1 needs the artefact and the CoCon that shall be checked for wether any of the artefact's elements violate it. In the 'identify constrained elements' loop, the algorithm searches for any artefact elements that fulfil the CoCon's context conditions. In the next loop 'check constrained elements', each possible pair of constrained elements is checked for wether it violates $C(x, y)$. In algo-

rithm 1, $Semantics^{C(x,y)}_{Artefact-Type}$ represents a condition on how to check to artefacts of a certain type whether the artefact elements x and y comply with $C(x, y)$. If the algorithm returns an empty result relation R then no conflicts were found. Otherwise, R contains the pairs of those elements which violate the CoCon. This result allows us to trace which element is inconsitent to which other element. As in the xlinkit framework (see section 3.5), this result could be expressed as hyperlinks between inconsistent elements.

The set of artefact elements A contains n elements. Hence, the 'identify constrained elements' loop runs $2n$ times. If each context condition check has linear complexity than the overall complexity for identifying the constrained elements is $O(n)$. If no constrained elements are identified then the Detect-CoCon-Violation algorithm ends without finding any CoCon-violation conflict. Thus, the **best case complexity** of algorithm 1 is $O(n)$ if checking the context conditions $TCC(x)$ and $SCC(x)$ has a linear complexity.

If the 'identify constrained elements' loop finds any constrained elements then the algorithm continues. Let s be the number of constrained scope set elements and t the number of constrained target set elements. The 'check constrained elements' loop runs $s \times t$ times. With each run, the loop evalutates whether the current pair of constrained elements fulfils the CoCon-predicate. The worst case complexity occurs if *all* n artefact elements are contained both in the scope set and in the target set of the CoCon. In that case, both s and t equal n – the 'check constrained elements' loop runs n^2 times. Hence, the Detect-CoCon-Violations algorithm has the **worst case complexity** of $O(n^2)$ if checking each pair of constrained elements for $Semantics^{C(x,y)}_{Artefact-Type}$ has linear complexity.

But, the complexity of algorithm 1 can grow out of hand for two reasons. On the one hand, checking the context conditions $TCC(x)$ or $SCC(x)$ can have high complexity if they refer to a complex context model and depending on the query capabilities or their query language. Hence, we suggest using context models and context conditions that result in linear complexity. In regards to context models, we do not consider the context of context (...of context) as explained in section 2.1 because queries to such a recursive context model might have undecidable complexity.

On the other hand, checking the constrained elements for wether they fulfil the CoCon-predicate can have high complexity if the CoCon-predicate is complex or if its artefact-type-specific $Semantics^{C(x,y)}_{Artefact-Type}$ are complex. In order to avoid complex CoCon-predicates, we propose using only *binary* relations that relate *two* elements. Using k-ary relations results in $O(n^k)$ complexity because every additionally related element calls for another nested loop within the 'check constrained elements' loop. But, even checking binary relations can have high complexity. A CoCon-violation conflict can only be detected automatically if the artefact-type-specific semantics of the checked CoCon-predicate are defined and computable.

5. Detect Contradicting CoCons

Section 4 has discussed CoCon-violation conflicts, where artefact elements violate a CoCon. This section will examine inter-CoCon conflicts, where one CoCon contradicts another CoCon.

5.1. Inter-CoCon Conflicts

An **inter-CoCon conflict** occurs if two CoCons contradict each other. This section discusses how to detect inter-CoCon conflicts to protect the requirements expressed via CoCons from unwanted violation. It defines general **inter-CoCon conflict types** that apply to two CoCons *of the same CoCon-predicate* $C(x, y)$.

The first inter-CoCon conflict type is called $NOP \leftrightarrow NOT$ conflict because it can occur if one CoCon has a `NOT` operation, while another CoCon of the same CoCon-predicate does not have a CoCon-predicate operation (no operation is abbreviated as **NOP** here).

$NOP \leftrightarrow NOT$ Inter-CoCon Conflict: The two CoCons

- $\forall x, y : TR(x) \wedge TCC_1(x) \wedge SR(y) \wedge SCC_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : TR(x) \wedge TCC_2(x) \wedge SR(y) \wedge SCC_2(y) \rightarrow \neg C(x, y)$

contradict each other if $\exists x, y : TCC_1(x) \wedge TCC_2(x) \wedge SCC_1(y) \wedge SCC_2(y)$.

If $C(x, y)$ is defined as `x MUST BE ACCESSIBLE TO y` CoCon-predicate then the $NOP \leftrightarrow NOT$ inter-CoCon conflict states that no element x must both be `ACCESSIBLE TO` and `NOT ACCESSIBLE TO` any y. For instance, following two CoCons can cause a $NOP \leftrightarrow NOT$ inter-CoCon conflict:

CoCon 1: The privacy policy of section 1.2 can be expressed in CLL as `ALL COMPONENTS WHERE 'Personal Data' = 'True' MUST NOT BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Create Report'` .

CoCon 2: `ALL COMPONENTS WHERE 'Personal Data' = 'True' MUST BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Human Resources'`

CoCon 1 contradicts CoCon 2 if any component exists that both belongs to the operational area 'Human Resources' and is used in the workflow 'Create Report'.

The next two inter-CoCon conflict types take the CoCon-predicate operation ONLY into account. A $NOP \leftrightarrow ONLY$ inter-CoCon conflict can occur if one CoCon without CoCon-predicate operation (=NOP) contradicts another CoCon with the CoCon-predicate operation ONLY:

$NOP \leftrightarrow ONLY$ inter-CoCon conflict: The two CoCons

- $\forall x, y : TR(x) \wedge TCC_1(x) \wedge SR(y) \wedge SCC_1(y) \rightarrow C(x,y)$ and
- $\forall x, y : TR(x) \wedge \neg TCC_2(x) \wedge SR(y) \wedge SCC_2(y) \rightarrow \neg C(x,y)$
  $\forall x, y : TR(x) \wedge TCC_2(x) \wedge SR(y) \wedge SCC_2(y) \rightarrow C(x,y)$

contradict each other if $\exists x, y : TCC_1(x) \wedge TCC_2(x) \wedge SCC_1(y) \wedge \neg SCC_2(y)$.

If $C(x,y)$ is defined as `x MUST BE ACCESSIBLE TO y` CoCon-predicate then the $NOP \leftrightarrow ONLY$ inter-CoCon conflict states that no element x must be ACCESSIBLE TO y if x is not ONLY ACCESSIBLE TO y. For instance, the following CoCons can cause a $NOP \leftrightarrow ONLY$ inter-CoCon conflict:

CoCon 3 : `ALL COMPONENTS WHERE 'Personal Data' CONTAINS 'True' MUST ONLY BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'`

CoCon 2 contradicts CoCon 3 if any component x handling personal data is accessible to a component y that is used in the human resources department but that is *not* used by the controlling department. In that case, CoCon 3 demands that x must not be accessible to y because y is not used by the controlling department. But, x must be accessible to y due to the CoCon 2 - an inter-CoCon conflict.

The 'NOT-SELF' inter-CoCon conflict occurs if the CoCon's scope set overlaps with the CoCon's target set and if the CoCon-predicate is reflexive ($\forall x : C(x,x)$).

'NOT-SELF' inter-CoCon conflict: The CoCon $\forall x, y : TR(x) \wedge TCC(x) \wedge TR(y) \wedge SCC(y) \rightarrow C(x,y)$ contradicts the proposition $\exists x : SCC(x) \wedge TCC(x)$.

ACCESSIBLE TO CoCons are reflexive, because a component always is ACCESSIBLE TO itself. Moreover, the target set and the scope set of ACCESSIBLE TO CoCons can overlap, because both sets can contain components. If $C(x,y)$ is defined as `x MUST BE ACCESSIBLE TO y` CoCon then the 'NOT-SELF' inter-CoCon conflict states that no element x must be NOT ACCESSIBLE TO itself.

CoCon 1 violates this inter-CoCon conflict if any component x handles personal data and is used in the controlling department. In that case, CoCon 1 demands that x cannot access itself. This is absurd. Hence, x must be changed until it *either* handles personal data *or* belongs to the controlling department . It may not belong to *both* contexts. If it is not possible to adjust the components accordingly then the system cannot comply with this requirement.

Besides the inter-CoCon conflicts discussed her, more inter-CoCon conflicts exist as listed in [1].

5.2. The Detect-Inter-CoCon-Conflicts Algorithms

This section discusses algorithms for finding inter-CoCon conflicts. First, the algorithm for detecting , NOP $\leftrightarrow$ NOT' inter-CoCon-conflicts is presented and its complexity is examined. Afterwards, more algorithms for detecting other inter-CoCon conflicts are discussed. As input, algorithm 2 needs an artefact and the two CoCons that shall be checked for wether they contradict each other. Both CoCons must have the same CoCon predicate $C(x,y)$. $CoCon_1$ has no CoCon-predicate operation, while the other $CoCon_2$ has a NOT ($\neg$) CoCon-predicate operation. The algorithm loops over all artefact elements and searches if any of them fulfil the condition defined in section 5.1 for $NOP \leftrightarrow NOT$ inter-CoCon-conflicts. As a result, the algorithm returns two sets $TARGET$ or $SCOPE$ which contain those elements that cause a conflict between $CoCon_1$ and $CoCon_2$.

The set of artefact elements A contains n elements. Hence, $4n$ context condition checks happen in the loop of algorithm 2. If each context condition check has linear complexity than the overall complexity of algorithm 2 is $O(n)$.

Algorithm 2 only covers the first inter-CoCon conflict listed in section 5.1. Each other type of inter-CoCon conflicts needs an algorithm of its own. These additional algorithms are not defined here because they are similar to algorithm 2 and can be generated from the inter-CoCon conflict definitions listed in section 5.1.

The performance of checking inter-CoCon conflicts can still be improved. Instead of having one simple algorithm for each inter-CoCon conflict, all inter-CoCon conflicts could be checked in one big algorithm that only loops *once* over all artefact elements in order to compute all context condition checks needed for any inter-CoCon conflict. Thus, the overall number of context condition checks for testing all inter-CoCon conflict types could be reduced. Moreover, the simple algorithms discussed here only check a pair of two CoCons. If more than two CoCons of the same CoCon-predicate exists the pair-checking-algorithms must be invoked several times for each each CoCon-pair combination. Within each invocation of a pair-checking-algorithm, the context conditions of both CoCons are checked for all artefact elements. Again, the total number of context condition checks could be reduced if those context condi-

input : The finite set A containing all n elements $a_1, ..., a_n$ of the checked artefact and the two CoCons $\forall x, y \in A : TR(x) \wedge TCC_1(x) \wedge SR(y) \wedge SCC_1(y) \rightarrow C(x, y)$ ($CoCon_1$) and $\forall x, y \in A : TR(x) \wedge TCC_1(x) \wedge SR(y) \wedge SCC_1(y) \rightarrow \neg C(x, y)$ ($CoCon_2$)

output : The set $SCOPE$ containing those scope set elements that cause a conflict between $CoCon_1$ and $CoCon_2$ and set $TARGET$ containing those target set elements that cause a conflict between $CoCon_1$ and $CoCon_2$

/* check $\exists x, y : TCC_1(x) \wedge TCC_2(x) \wedge SCC_1(y) \wedge SCC_2(y)$ */
foreach $a_i \in A$ **do**
  if $SCC_1(a_i)$ *and* $SCC_2(a_i)$ **then**
    add a_i to the set $SCOPE$
  if $TCC_1(a_i)$ *and* $STCC_2(a_i)$ **then**
    add a_i to the set $TARGET$
/* conflicts only exist if both result-sets are not empty */
if $TARGET$ *is empty OR* $SCOPE$ *is empty* **then**
  return *two empty sets*
else
  return *the two sets* $TARGET$ *and* $SCOPE$

Algorithm 2: Detect-$NOP \leftrightarrow NOT$ Inter-CoCon-Conflicts

tion checks done in previous CoCon-pair-checks would not be repeated. However, the goal of this section is not to develop the fastest solutions for specific applications in detail. All in all, the average complexity will always be $O(n)$ because each solution must iterate over all artefact elements. Instead, the goal of this section is to provide an general detect-inter-CoCon-conflicts algorithm in order to discuss in which way it differs from the general detect-CoCon-violations algorithm presented in section 4.2.

Both the Detect-CoCon-Violations algorithm (No 1) and the Detect-Inter-CoCon-Conflicts algorithm (No 2) have one common limitation: they both need the artefact elements in order to be run. It is not possible to detect CoCon-violations without artefact elements, and it is not possible to detect inter-CoCon conflicts without artefact elements. The first difference is obvious: algorithm 1 has an average complexity of $O(n^2)$, while algorithm 2 has $O(n)$. Detecting inter-CoCon conflicts is faster than detecting CoCon-violation conflicts. But, an even more important difference exists: algorithm 1 needs artefact-specific semantics, while algorithm 2 doesn't. In algorithm 1, $Semantics^{C(x,y)}_{Artefact-Type}$ represents a condition on how to check to artefacts of a certain type whether the artefact elements x and y comply with $C(x, y)$. Hence, algorithm 1 has two limitations:

On the one hand, checking the artefact-specific $Semantics^{C(x,y)}_{Artefact-Type}$ for each pair of constrained elements reduces the overall performance of algorithm 1. On the other hand, algorithm 1 simply cannot be run if $Semantics^{C(x,y)}_{Artefact-Type}$ is unkown. And it can compute wrong results if $Semantics^{C(x,y)}_{Artefact-Type}$ is incomplete or wrong. On the contrary, algorithm 2 does not depend on the artefact-specific $Semantics^{C(x,y)}_{Artefact-Type}$. It doesn't get slower if $Semantics^{C(x,y)}_{Artefact-Type}$ has a high complexity. It doesn't compute wrong results if $Semantics^{C(x,y)}_{Artefact-Type}$ is incomplete or wrong. And finally, it can be run even if $Semantics^{C(x,y)}_{Artefact-Type}$ is undefined. A software system consists of many different artefact types. Even if the artefact-specific semantics of some artefact type used in the system are undefined algorithm 2 still can detect inter-CoCon-conflicts and, thus, identify contradicting requirements.

6. Proof-of-Concept Tools

According to [14], traceability makes it feasible to examine the whole set of objects and links for a project and thus permits conflict detection and helps to ensure that decisions made later in the process are consistent with earlier decisions. The previous sections have explained how to detect both CoCon-violation conflicts and inter-CoCon conflicts. If a design decision is expressed via a CoCon than a tool can automatically protect it in later design decisions. Thus, CoCon can prevent unwanted 'corrections' because they define invariants whose violation can automatically be detected in a modification.

Tools can automatically identify the elements constrained by CoCons and notify the designer if an element violates a CoCon or if a CoCon contradicts another CoCon. Two prototypical monitoring tools for different artefact types have been developed:

- The open source CASE tool 'ArgoUML' already has a model-validation mechanism called design critics ([16]). The prototypical CCL-plugin ([1]) adds design critiques to the open source CASE tool ArgoUML that check the compliance of a UML model with CoCons.

Most of the CoCons mentioned in this article express access permissions and refer to components. Instead, section 1.3 demonstrates a CoCon that expresses an availability requirement refers to data and computers. The CCL-plugin for ArgoUML can both validate security and availability CoCons.

- The prototypical 'EJB-Complex' framework presented in [12] can validate Enterprise Java Beans for compliance with CoCons at runtime. It uses 'ECA plugin templates' for defining the artefact-type-specific semantics of CCL for EJB systems. It generates ECA-rules (event condition action rules) from a CoCon, intercepts method invocations and uses these rules to control the relevant method invocations. For instance, it can control which component can invoke which other component according to the current context of the components.

Additionally, CoCons are used by the pharmaceutical web application 'dd-pro' (see www.imphar.com). It is a document management system. For instance, CoCons define who can access which document according to the document's operational area and the user's role and the user's current client. However, the details about applying CoCons in document or data management are currently not published yet.

7. Conclusion

Requirements traceability is intended to ensure *continuous* alignement between stakeholder requirements and system evolution: after each modification, all the system artefacts should be checked for whether their elements still comply with all the requirements. Limited ressources often prevent achieving absolute consistency. But, tools that facilitates to detect which artefact elements don't comply with which requirements helps to improve consistency.

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.

Within one system, only one ontology for metadata 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. We recommend using international standard ontologies. For instance, at imphar AG we express context via the MedDra dictionary. It defines the symptoms of a patient on different levels and is used by most pharmaceutical authorities and companies on this planet. If more than one ontology for metadata is used, correspondences between heterogeneous context property values can be expressed via inter-value CoCons ([1]) or model correspondence assertions ([4]).

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 requirements affecting several elements that are not associated with each other or even belong to different artefacts.

Algorithms for automatically detecting both violated and contradicting CoCons have been presented. 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. Inter-CoCon conflicts can even be detected if the precise semantics of the checked system artefact are unkown.

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 constraint which is much longer and much more complicated 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 requirement 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 identifiy 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 are affected by the 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.

Maintenance is a key issue in continuous software engineering. 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 requirements.

References

[1] F. Bübl. The context-based constraint language CCL for components. Technical Report 2002-20, Technical University Berlin, Germany, available at www.CoCons.org, October 2002.

[2] F. Bübl. Introducing context-based constraints. In H. Weber and R.-D. Kutsche, editors, *Fundamental Approaches to Software Engineering (FASE '02), Grenoble, France*, volume 2306 of *LNCS*, pages 249–263, Berlin, April 2002. Springer.

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

[4] S. Busse. Modellkorrespondenzen für die kontinuierliche Entwicklung mediatorbasierter Informationssysteme. PhD Thesis, Technical University Berlin, Germany, Logos Verlag, 2002.

[5] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic, Boston, 2000.

[6] N. Damianou. A policy framework for management of distributed systems. PhD Thesis, Imperial College, London, UK, 2002.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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, 5th *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[13] C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *International Conference on Automated Software Engineering (ASE)*, Coronado Bay, CA, 2001.

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

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

[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] S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.

[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] M. Sloman and K. P. Twidle. Domains: A framework for structuring management policy. In M. Sloman, editor, *Chapter 16 in Network and Distributed Systems Management*, pages 433–453, 1994.

[20] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In 5th *IEEE International Symposium on Requirements Engineering, Toronto*, pages 249–263. ACM Press, August 2001.