Tracing Crosscutting Requirements for Component-Based Systems via Context-Based Constraints

vorgelegt von
Diplom-Informatiker
Felix Bübl

an der Fakultät IV Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr. Ing. -

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Peter Pepper
Berichter:	Prof. Dr. Herbert Weber
Berichter:	Prof. Dr. Heinrich Hussmann

Tag der wissenschaftlichen Aussprache: 22. Juni 2005

Berlin 2005
D 83

Abstract

We often fail to keep track of requirements in complex software systems because we cannot cope with all the details - in particular, it is expensive to check a system for compliance with crosscutting requirements where one requirement affects several parts of the system. In order to detect requirement violations, each system element involved in a requirement must be identified and checked for whether it meets the requirement. But, it is difficult to identify which system element is involved in which requirement in complex or frequently changing software systems. In this thesis, I specify crosscutting requirements via constraints in order to automatically check the system for compliance and to automatically identify contradicting requirements. I present a new notion of constraints: a context-based constraint (CoCon) expresses a condition on how its constrained elements must relate to each other. CoCons are *adaptive* in order to cope with complex system: A CoCon can identify the system elements affected by the requirement automatically because it can *indirectly* select its constrained elements according to their context. Five different CoCon families for component-based systems are discussed: 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. Distribution CoCons express which components must (or must not) be allocated to which computers. Information-Need CoCons express which users must (or must not) be notified of which documents. Finally, Inter-Value CoCons express whether an element in a certain context must (or must not) reside in another context. This thesis focuses on applying CoCons in UML models of component-based systems.

I present algorithms for detecting both violated and contradicting CoCons automatically. Inter-CoCon conflicts can even be detected if the precise semantics of the checked system artefact are unknown. Moreover, CoCons support the design of software systems from the start of the development process. In contrast to OCL constraints, CoCons specified during modelling can already be checked during modelling at the same metalevel. Hence, the model can be checked for violated or contradicting CoCons already during modelling.

CoCons enable us to handle crosscutting requirements for possibly large, overlapping and dynamically changing sets of system elements - even across different artefact types or platforms. Writing down a requirement directly for each individual element involved in each system artefact is expensive in complex systems. Instead, a CoCon automatically constrains those elements whose context properties match with the CoCon's context condition. Hence, CoCons facilitate checking large-scale or frequently changing systems for compliance with crosscutting requirements during (re-)design, during (re-)configuration, and at runtime.

Zusammenfassung

Bei der Anpassung eines Softwaresystems an neue Anforderungen sollten bestehende Anforderungen nicht versehentlich verletzt werden. Bei komplexen oder sich häufig ändernden Systemen verlieren wir jedoch leicht den Überblick darüber, welches Systemelement welche Anforderung umsetzt. Besonders unübersichtlich ist die Handhabung von Querschnittsanforderungen, bei denen mehr als ein Systemelement von einer Anforderung betroffen ist.

Diese Dissertation schlägt eine neue Art von Constraints vor, um Querschnittsanforderungen (engl: Crosscutting Concerns) ausdrücken und ihre Einhaltung überwachen zu können: ein Context-Based Constraint (CoCon) kann die betroffenen Elemente des Systems anhand ihrers Kontextes erkennen. Die hier definierte Context-Based Constraint Language CCL besteht aus 22 verschiedenen Arten von CoCons zur Beschreibung von Anforderungen an komponenten-basierte Systeme in folgenden Bereichen: Access Permission CoCons beschreiben, wer auf was (nicht) zugreifen darf. Distribution CoCons legen fest, welche Komponenten (nicht) auf welchen Rechnern verfügbar sind. Information Need CoCons spezifizieren, wer über was (nicht) informiert wird. Communication CoCons beschreiben, auf welche weise Aufrufe zwischen Komponenten (nicht) erfolgen. Und schlieślich definieren Inter-Value CoCons Abhängigkeiten zwischen den Context Property Werten eines Elements.

Im Rahmen einer Softwareentwicklung entstehen verschiedene Artefakt, wie etwa Modelle, Source Code oder Laufzeitkomponenten. CoCons formulieren abstrakte Anforderungen unabhängig von Artefakt-Typen. Daher ermöglichen es CoCons, Softwaresysteme sowohl während ihrer Modellierung, während ihrer Konfiguration als auch zur Laufzeit auf die Einhaltung von Anforderungen zu überwachen. Diese Dissertation beschränkt sich auf die Anwendung von CoCons während der Modellierung mit UML.

Algorithmen zum Entdecken von verletzten oder widersprchlichen CoCOns werden vorgestellt. Anders als bei OCL Constraints kann ein Modell auf die Einhaltung von CoCon überwacht werden, ohne das im Modell spezifizierte System zu starten oder zu simulieren. Daher knnen mit CoCons schon frh im Entwicklungsprozess Systeme auf die Einhaltung von Querschnittsanforderungen geprft werden.

Bisherige Constraints werden direkt an die betroffenen Element annotiert. Es ist aufwendig, eine viele Elemente betreffende Querschnittsanforderung an jeder zugehörigen Stelle zu annotieren. Im Gegensatz dazu werden die von einem CoCon betroffenen Elemente indirekt anhand ihrer Context Properties identifiziert. Dadurch kann ein CoCon eine Anforderungen für möglicherweise zahlreiche, sich häufig verändernde Elemente beschreiben - sogar über Plattform- oder Modellgrenzen hinweg. Daher eignen sich CoCons insbesondere dafür, grośe oder sich häufig ändernde Systeme während der Modellierung, der Konfiguration oder zur Laufzeit auf die Einhaltung von Anforderungen zu überwachen.

Acknowledgments

First of all I thank my two supervisors for their support and advice during the preparation of this thesis: Heinrich Hussmann particularly improved my work through numerous valuable comments on technical and structural issues that were crucial for editing the drafts into the final dissertation. Herbert Weber supported my research by employing me. I always enjoyed the lively and controversial discussions with my colleagues at the Technical University of Berlin. Martin Grosse-Rhodes insightful comments on formalization and presentation helped me in improving the text's maturity a lot. Ralf-Detlef Kutsche gave me academic guidance and many thought-provoking conversations that helped me a great deal. Without Andreas Leicher there would not be a dissertation, only a collection of ideas. From the start on, he became my bug exorcist who identified missing, superfluous or wrong details. Moreover, he helped me in developing two proof of concept tools. Markus Schaal assisted me in formalizing my concepts from the start on when my ideas were still fuzzy. The discussions and publications with Michael Balser, Susanne Busse, Thomas Kabisch, Ulrich E. Kriegel, and Thomas Off inspired and encouraged me. By coaching my diploma thesis, Jürgen Oheim helped me entering the scientific community. Barbara Petkus and Valerie Adelsberger not only proofread multiple versions of my texts, but also provided many stylistic suggestions that improved my presentation and clarified my arguments.

Among the many the students working on the CCL-plugin for the case tool ArgoUML, especially Martin Skinner, Marek Feldo, Ute von Angern, Camara Lenuseni, and Priscilla-Mumene Nkweti put much effort in getting it up and running when writing their diploma thesises. Likewise, the EJBcomplex framework developed in the diploma thesises of Alexander Bilke and Jianxin Wang helped me to improve my concept. As well, both Frank Ratzlow work on aspect-oriented approaches and Joseph Bauer methodical discussions helped me improving my thesis.

Contents

1 Background **10**

1.1 Component-Based Software Engineering 10

1.2 The Unified Modelling Language UML 11

1.3 Requirements Engineering 12

1.4 The Object Constraint Language OCL 14

2 Outline of this Thesis **16**

2.1 Motivation: Adapting Complex Systems to New Requirements 16

2.2 The Running Example: A Privacy Policy 17

2.3 Objectives 17

2.4 Structure of this Thesis 19

3 Context-Based Constraints (CoCons) **21**

3.1 Overview 21

3.2 Introducing Context Properties 22

3.2.1 What is Context? 22

3.2.2 Context Properties: Formatted Metadata Describing Elements 24

3.2.3 Context Property Examples 25

3.2.4 Formal Definition of Context Properties 26

3.2.5 Useful Context Property (Stereo-)Types 27

3.2.6 Research Related To Context Properties 28

3.3 Introducing Context-Based Constraints (CoCons) 30

3.3.1 Intuitive Definition of Context-Based Constraints 30

3.3.2 The Common CoCon Syntax 33

3.3.3 The Context Property Query Language CPQL . 35

3.3.4 Navigation via Dot-Path-Notation 38

3.3.5 Two-Step Approach for Defining CoCon-Predicate Semantics 39

3.3.6 Formalization of Context-Based Constraints . . . 40

3.3.7 Formalization of CPQL 43

3.3.8 Comparing Context-Based Constraints with Aspects 43

3.3.9 Research Related to Context-Based Constraints . 46
3.3.10 The Fundamental Things Apply As Time Goes By 48
3.4 Turning CoCons into CoCon-Rules by Adding Events And Actions 49
3.4.1 Introduction to Business Rules and Policies 49
3.4.2 Difference between CoCons and CoCon-Rules . . 49
3.4.3 Limitations of Enriching CoCons with Actions and Events 50
3.4.4 Referring to Events and Actions in CoCon-Rules 51
3.4.5 The Common CoCon-Rule Syntax 51

4 Applying CoCons in Continuous Software Engineering 53
4.1 Continuous Requirements Tracing 53
4.2 Detect CoCon Violations 55
4.2.1 CoCon-Violation Conflicts 55
4.2.2 The Detect-CoCon-Violations Algorithm 56
4.3 Detect Contradicting CoCons 57
4.3.1 Inter-CoCon Conflicts 57
4.3.2 The Detect-Inter-CoCon-Conflicts Algorithms . . 60
4.4 Proof-of-Concept Tools 62
4.5 Maintaining Context Property Values 62
4.5.1 Type-Instance Constraint On Context Property Values 63
4.5.2 Dependent Context Property Values 64
4.5.3 Belongs-To Relations Result in Derived Context Properties Values 66
4.5.4 Outlook: Applying Context Properties in Continuous Software Engineering 69

5 The Context-Based Constraint Language CCL 71
5.1 Overview on CCL 71
5.2 Access Permission CoCons 71
5.2.1 The Notion of Access Permission CoCons 71
5.2.2 Access Permission CoCon-predicates 71
5.2.3 Detectable Inter-CoCon Conflicts of Access Permission CoCons 72
5.2.4 Example for Using Access Permission CoCons . . 72
5.2.5 Related Research on Access Control Policies 72
5.3 Communication CoCons 73
5.3.1 The Notion of Communication CoCons 74

5.3.2 The Communication CoCon-predicates 74
5.3.3 Detectable Inter-CoCon Conflicts of Communication CoCons 74
5.3.4 Examples for Using Communication CoCons . . . 75
5.3.5 Related Research on Communication CoCons . . . 75
5.4 Distribution CoCons 75
5.4.1 The Notion of Distribution CoCons 75
5.4.2 Distribution CoCon-predicates 76
5.4.3 Detectable Inter-CoCon Conflicts of Distribution CoCons 76
5.4.4 Examples for Using Distribution CoCons 77
5.4.5 Related Research on Distribution and Network Policies 77
5.5 Information-Need CoCons 78
5.5.1 The Notion of Information-Need CoCons 78
5.5.2 The Information-Need CoCon-predicates 78
5.5.3 Detectable Inter-CoCon Conflicts of Information-Need CoCons 79
5.5.4 Examples for Using Information-Need CoCons . . 80
5.5.5 Related Research on Information-Need CoCons . 81
5.5.6 Information-Need CoCon-Rules 82
5.6 Inter-Value CoCons 83
5.6.1 The Notion of Inter-Value CoCons 83
5.6.2 The Inter-Value CoCon-predicates 83
5.6.3 Detectable Inter-CoCon Conflicts of Inter-Value CoCons 84
5.6.4 Examples for Using Inter-Value CoCons 85
5.6.5 Related Research on Inter-Value CoCons 85

6 UML-Specific Semantics of CCL **87**
6.1 Integrating CoCons into UML 87
6.1.1 The Easy Part: Using UML's Constraints and Tagged Values 87
6.1.2 The Problem: UML Constraints Don't Consider Point Cuts 88
6.2 Comparing Context-Based Constraints with OCL 89
6.2.1 UML Semantics of `ACCESSIBLE TO` CoCons 89
6.2.2 CoCons can be Verified Already at the Same Meta-Level 91
6.2.3 CoCons can Constrain Unassociated Elements . . 92

7 The CCL Analysis & Specification Method CCLM 93

7.1 Overview on CCLM . . . 93

7.1.1 Background: Adopting Kotonya's and Sommervilles's Requirements Engineering Process . . . 93

7.1.2 Introducing the 11 Activities during the CCLM Process . . . 95

7.1.3 Preamble: Why to Apply CCL at all? . . . 97

7.2 The Requirements Elicitation Phase of CCLM . . . 98

7.2.1 Rules-Driven or Context-Driven Requirements Elicitation . . . 98

7.2.2 How to Write Down Informal CoCon-Rules . . . 99

7.2.3 Rule-Driven CoCon Family Identification . . . 101

7.2.4 Rule-Driven Elicitation of Relevant Informal CoCon-Rules . . . 101

7.2.5 Rules-Driven Elicitation of Relevant Context Property Candidates . . . 104

7.2.6 Context-Driven Elicitation of Relevant Context Properties . . . 104

7.2.7 Context-Driven Elicitation of Relevant Informal CoCon-Rules . . . 105

7.2.8 Context-Driven CoCon Family Identification . . . 106

7.3 The Requirements Negotiation Phase of CCLM . . . 108

7.4 The CCL Specification Phase of CCLM . . . 109

7.4.1 Context Property Application . . . 109

7.4.2 CCL Specification . . . 111

7.5 The CCL Validation Phase of CCLM . . . 112

8 Conclusion 114

8.1 Summary . . . 114

8.2 Future Research Recommendations . . . 115

8.3 Limitations of CoCons . . . 116

8.4 Benefits of CoCons . . . 117

A The Textual Syntax of CCL 119

B Translating Accessability CoCons into OCL 121

B.1 Overview . . . 121

B.2 Artefact-Specific Semantics for UML . . . 124

B.3 The Privacy Policy in OCL . . . 131

Bibliography 135

List of Figures

2.1 Towards Loosely Coupled Software Systems 17

3.1 Graphical Notation for Context Properties 25
3.2 The CoCon Metamodel 31
3.3 A CoCon Relates any Element of the 'Target Set' with any Element of the 'Scope Set' 31
3.4 The Replaceable Conceptual Modules of CoCons 33
3.5 Two-Step Approach for Defining the Semantics of CoCon-Predicates 39

4.1 Excerpt of a Call Graph Depicting Dependencies Between 15 Classes 54
4.2 Context Property Values Can Depend On The Current Configuration 66

6.1 A Point Cut indirectly associates a Constraint with its Constrained Elements 87
6.2 The Component 'A' is Not Allowed to Access the Component 'B' 89

7.1 Overview on the Four Phases of CCLM for Identifying Context-Based Business Rules 94
7.2 Iterative Refinement of CCL Constraints via CCLM . . . 95
7.3 CCLM consists of 11 Activities 96
7.4 The WinWin Negotiation Model 108

B.1 A Component Diagram Showing Component Types that violate the 'Privacy Policy' CoCon 121
B.2 UML 2.0 Metaclasses For Component Types in Component Diagrams 122
B.3 Metaclass Component with a Virtual OCL Operation . . 123

1. Background

This chapter explains well-known technical terms used in this thesis. First, component-based software systems are described in section 1.1. Afterwards, requirements engineering is outlined in section 1.3. Next, the Unified Modelling Language UML is sketched in section 1.2. Finally, the Object Constraint Language OCL is explained in section 1.4.

In this document, **new terms** are highlighted boldly where they are explained.

Quick Tour $\xrightarrow{goto}$ Section 2.4

Experienced readers who already know the basic terms of components, requirements engineering and UML can skip the next pages and continue in chapter 2.4 on page 19, where the contribution of this thesis is outlined.

1.1 Component-Based Software Engineering

Many Definitions

According to [Som92], the specification, development, management and evolution of software systems makes up the discipline of **software engineering**. In traditional software engineering, a software system is developed as one monolithic block. Component-based software engineering replaces this monolithic approach with assembling software systems from software components. This thesis focuses on component-based software systems. The basic terms of component-based software engineering are explained next. More details are, e.g., provided in [CHJK02, Pal02, SP96]. Many different definitions of the term **component** exist in software engineering. The approach presented in this thesis does not stick to one of them. Instead, it can be adapted to each definition of component as discussed in section 3.3.5. Some popular definitions are discussed next:

[Szyperski]

> “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties ” ([SP96])

[Bachman]

> “A component is a software implementation that can be executed on a physical or logical device. A component implements one or more interfaces that are imposed upon it. This reflects that the component satisfies certain obligations, described as a contract. These contractual obligations ensure that independently developed components obey certain rules so that components interact (or can not interact) in predictable ways, and can be deployed into standard build-time and run-time environments.” ([BBB$^+$00])

Both Bachman and Szyperski consider the interfaces as contracts between the components. Interfaces are a key aspect of components. An interface defines how to interact with a component, but hides the underlying implementation details. The more we know about a component’s interfaces, the better we can handle this component. A component specification should assist clients and designers in understanding what services are offered by the component, and allow them to match the component’s capabilities to their specific needs.

Focus: Design

There are a number of component specification techniques ranging from formal (using mathematical notations) to non-formal (using a natural language description). However, there is no standard method for specifying components. This thesis focuses on designing component-based systems via the standard modelling language UML, which is outlined in the next section.

1.2 The Unified Modelling Language UML

Modelling

As a result of software engineering activities, **artefacts** are created. For example, source code, configuration files, or models are artefacts created when developing a software system. Modelling is the designing of software applications before coding. A **model** plays the analogous role in software development that blueprints and other plans play in the building of houses. Many languages for modelling software systems exist. This section outlines the basic features of the currently most popular modelling language.

Development 'Level'

The distinction of the software development process into different *phases* is outdated. On the contrary, incremental and iterative software development approaches switch between analysis level, design level, implementation level and runtime level during each iteration. Therefore, the term 'design phase' is not used here. Instead, the terms 'during modelling' or 'at modelling level' or 'in the model' are used.

UML

The **Unified Modelling Language (UML)** has emerged as the software industry's dominant language for modelling object-oriented software systems. UML consists of twelve diagram types that assist developers in specifying, visualizing, and documenting models of software systems. Different versions of UML exist. Currently, some UML tools support version 1.3, some 1.4 and some 2.0. The Object Management Group (OMG) adopted UML as its standard modelling language. The OMG is a consortium that produces and maintains computer industry specifications for interoperable enterprise applications. It is proposing the UML specification for international standardization.

UML Components

The UML became the modelling standard for modelling object-oriented software systems just at the time that components were starting to move into the mainstream. It supports components as implementation concepts but provides no explicit support for other aspects of their specification. Hence, a number of UML enhancements for modelling component-based systems have been proposed. This thesis sticks to the 'UML components' approach introduced in [CD00]. The UML components approach, however, ignores the composition of components as the method of assembling software components to composed software systems. Therefore, composition is additionally addressed in the relevant sections of this thesis.

UML Metamodel

A UML diagram contains symbols. For example, a class diagram can contain rectangle symbols that represent classes. In order to understand UML diagrams, the semantics of each symbol must be defined. A **metamodel** is a precise definition of the constructs and rules needed for creating semantic models. For example, it defines the semantics of rectangle symbols that represent classes. The UML metamodel defines the meaning of each UML symbol - it exactly describes the concepts covered in UML.

The UML metamodel definition consists of text and diagrams. The diagrams used to define UML concepts are UML diagrams themselves. To illustrate this dichotomy closer, consider the following analogy: A book describing the grammar of the English language can be written in English. In this case, it would be describing valid constructs of the English language and the semantics of these constructs, while using the same language, it is describing to convey the description.

UML architecture

According to [OMG03b], the UML metamodel is defined as one of the layers of a four-layer metamodeling architecture:

Layer	**Scope**	**Defines**	**Sample constituents**
Meta-Metamodel	The infrastructure for a metamodeling architecture	Defines the language for specifying metamodels.	MetaClass, MetaAttribute, MetaOperation
Metamodel	An instance of a metametamodel.	The UML metamodel specifies the semantic of the UML model elements	Class, Attribute, Operation, Component
Model	An instance of a metamodel.	Defines a language to describe an information domain.	LinkedList, currentState, setValue()
Objects or Data	An instance of a model at runtime	Defines a specific information domain	[Person: name = Rolf, height = 183], 45.1, "hello"

The difference between metamodel and model is more closely discussed in chapter 6.

1.3 Requirements Engineering

Pamela Zave provides the following definition of requirements engineering in [Zav97]:

[Zave]

> "Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families."

This definition is attractive for a number of reasons. First, it highlights the importance of 'real-world goals' that motivate the development of a software system. These represent the 'why' as well as the 'what' of a system. Second, it refers to 'precise specifications'. These provide the basis for *analysing* requirements, *validating* that they are indeed what stakeholders want, *defining* what designers have to build, and *verifying* that they have done so correctly upon delivery. Finally, the definition refers to 'evolution over time and across software families', emphasising the reality of a changing world and the need to reuse partial specifications, as engineers often do in other branches of engineering.

[IEEE]

In [oEE90], the IEEE defines a requirement as:

1. A condition or capability needed by a user to solve a problem or achieve an objective.

2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in (1) or (2)

Better Fix Errors Early

Produced by Software Engineering Institute (SEI), the CHAOS report ([SG95]) reveals the following figure based on the investigation of over 7500 IT projects: 73 percent of projects are cancelled or fail to meet expectations due to poor requirements definition and analysis. Other studies, e.g. [Dav90], result in similar figures: the cost of correcting an error increases by an order of magnitude in later development stages. Hence, errors should be identified and fixed early.

Crosscutting Requirements

Typically, two classes of requirements are distinguished: **functional requirements** specify a function that the software system must be capable of performing . On the contrary, **non-functional requirements** describe issues such as performance, reliability, efficiency, usability, portability, testability, understandability or modifiability. Non-functional requirements tend to be crosscutting. A **crosscutting** requirement violates the separation-of-concerns paradigm: it is not possible to handle this requirement at only one single, encapsulated place in the system. Instead, it involves more than one system element.

Requirements Engineering Process

According to [KS98], the requirements engineering process includes the following activities:

- **Requirements Elicitation** is the activity during which software requirements are discovered, articulated, and revealed from stakeholders or derived from system requirements. Sources may be system requirements documentation, customers, end-users, and domain specialists or market analysis.
- **Requirements Analysis and Negotiation** is the activity during which the requirements gathered during elicitation are analysed for conflicts, ambiguity, inconsistencies, missing requirements or extra requirements. During this activity, negotiation between all stakeholders occurs to arrive at a set of agreed upon requirements.
- **Requirements Specification** is the activity during which the requirements are recorded in one or more forms, usually in a software requirements specification document. The requirements may be in natural language, a formal language or in a graphical form. The specification is used to communicate the requirements to customers, end-users, managers, designers, and system developers.
- **Requirements Validation** is the activity during which the requirements are checked for omitted, extra, wrong, ambiguous and inconsistent requirements This activity also checks to ensure that all requirements follow stated quality standards.

Traceability

Traceability is defined in [RJ01] as the ability to discover the history of every feature of a system. It determines how easy it is to read, navigate, query and change requirements documentation. Gotel defines requirements traceability in [GF94] 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 is a means of achieving integrity and completeness of that documentation, and has an important role to play in managing change.

Maintainability

As summarized in [NE00], it has long since been established that requirements management needs to be done throughout a system's lifetime. A large number of definitions of qualities exist that are related to the ability of a software system to be modified. An early definition is given in [MRW77]: **Maintainability** is the effort required to locate and fix an error in an operational program. Flexibility is the effort required to modify an operational program. A more recent classification of qualities is given in the ISO 9126 standard ([ISO00]): maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification. Although different in wording, the definitions are almost identical in their semantics.

Software Evolution

Software evolution is frequently used as another expression for **software maintenance**. According to [Ste01], a common interpretation of software maintenance used is to span the phase after first delivery of a product. The split into a development phase and a maintenance phase is problematic. It derives from the waterfall model described in [Som92]. In contrary to development processes in other disciplines of engineering, modern software development processes do not pass through these phases sequentially one after the other. Instead, the need to validate requirements and design forces a development team to incrementally pass through these phases over and over again. It has become evident that the development phase and the maintenance phase cannot be clearly separated. Development always incorporates also maintenance since a software system will never be mature after first delivery. A running system, on the other hand, will always have to be developed further to cope with changing requirements. The term **software evolution** incorporates both the software development and maintenance process in one expression.

1.4 The Object Constraint Language OCL

Recording Requirements

In complex applications, even experienced architects need tool support for designing and maintaining the system. For instance, they need to be reminded which of the requirements apply to which parts of the system. A prerequisite for keeping track of the requirements is writing them down. Using natural language for requirements specification introduces freedom of misinterpretations and gives tools no chance to cope with it. On the contrary, formal languages have precise semantics and can automatically be parsed by software tools. However, the language for expressing requirements must be comprehensible for designers, programmers and customers.

Software evolution is a major challenge to software development. When adapting a system to new, altered, or deleted requirements, existing requirements should not unintentionally be violated. By specifying a requirement as an invariant, it can be considered and protected in later modifications. Typically, UML invariants are specified via the Object Constraint Language OCL outlined next.

Constraint

According to [WK99], a **constraint** is a restriction on none or more values of (part of) an object-oriented model or system. In UML ([OMG03b]), a constraint is a semantic condition or restriction expressed which must be true for the model to be well formed. Constraints can be expressed in a language specially designed for writing constraints, a programming language, mathematical notation, or natural language.

OCL According to [CKM+99a, WK99, OMG03b], the **Object Constraint Language (OCL)** is a textual specification language, designed especially for the use in diagrammatic specification languages such as the UML. When regarding the UML diagrams as a language, it turns out that the diagram-based UML is limited in its expressiveness. OCL is deeply connected to UML diagrams, and can define textual constraints for UML model elements.

Origins OCL is a specification language that tries to mediate between the practical users needs and the theoretical work that has been done in that area. In particular, much of the theoretical work was done in the areas of algebraic specification languages, such as SPECTRUM [BFG+93], ACT ONE/TWO [EM90], or TROLL (*light*) [CGH92], model based specification techniques, such as Z [Spi88], and also data base query languages, such as SQL. The key ideas from these areas, such as navigation expressions or container types as abstract data types, have been taken and combined into a language that became part of the UML standard. According to [Rum98], several degrees of formality of a notation exist. OCL does currently not have a formally defined meaning and can therefore only be regarded as semi-formal. Due to the tight connection of OCL with the UML diagrams, the definition of a formal syntax for OCL must be based on a formal semantics for the UML.

2. Outline of this Thesis

Structure of this Chapter

After explaining important software engineering problems in section 2.1 and giving a short example in section 2.2, section 2.3 outlines how I propose to solve these problems in this thesis.

2.1 Motivation: Adapting Complex Systems to New Requirements

Even in simple systems, compliance with the requirements isn't achieved easily because each part of the system must be checked if it complies with the requirements. Validating compliance with the requirements becomes even more expensive if the system artefacts or the requirements are changed often because a lot of details must be checked *often* here. Every new, removed, or changed part of the system must be checked for compliance with every existing requirement. Modern software technologies increase this problem because they allow us to develop quickly changeable software systems as depicted in figure 2.1: batch data transfers have traditionally been accomplished through nightly magnetic tape inputs in the seventies. Recently, replaceable components have been used to provide a service-oriented interface to external systems. Nowadays, business-to-business web transactions are becoming loosely coupled according to [SvdH02]. Web services, message brokers, grid computing, or peer-to-peer networks are emerging communication technologies that facilitate the development of loosely coupled, large-scale software systems handling frequently changing data, consisting of frequently changing components deployed on frequently changing computers in frequently changing contexts. We often fail to keep track of requirements in such loosely coupled, complex software system because we cannot cope with all the quickly changing details.

A crosscutting requirement is a requirement that affects more than one system element. Requirements tend to change often. It is especially difficult to adapt complex systems to changes of crosscutting requirements because each modification must consider which of the frequently changing data or which of the frequently changing components deployed on frequently changing computers in frequently changing contexts are affected by which requirements. In this thesis, I focus on crosscutting requirements because we don't have a satisfying way to deal with them yet. Instead, we often fail to keep track of crosscutting requirements.

No single person has all the knowledge needed to design and maintain a complex system in every detail. Instead, teams of stakeholders providing some of the necessary knowledge and their own goals and priorities develop most complex systems. This 'thin spread of application domain knowledge' has been identified in [CKI88] as a general problem in software development. It impedes requirements validation because we cannot check the system without knowing which requirement affects with part(s) of the system. Typically, we try validating complex or frequently changing systems via automated tests: we write additional software that automatically checks the system for compliance with the requirements. But, each automated test can become outdated with every changed requirement, system element, or context. If we do not precisely know which

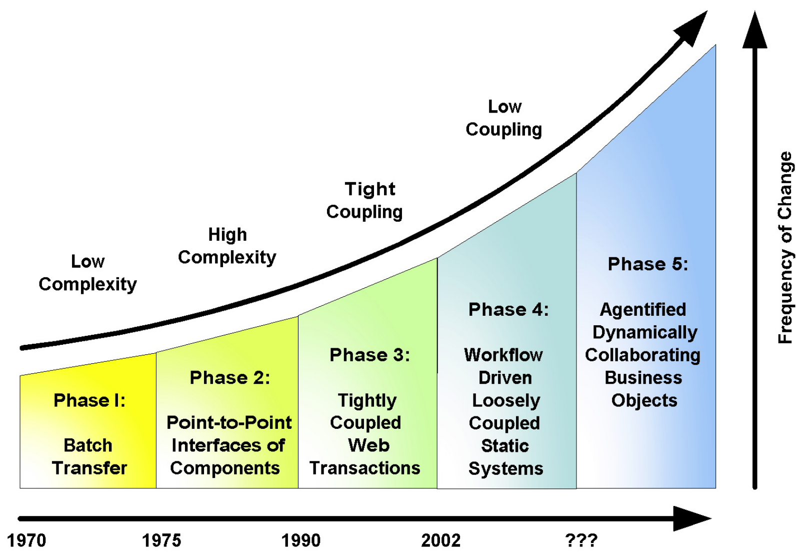
Low Complexity
High Complexity
Tight Coupling
Low Coupling
Phase I: Batch Transfer
Phase 2: Point-to-Point Interfaces of Components
Phase 3: Tightly Coupled Web Transactions
Phase 4: Workflow Driven Loosely Coupled Static Systems
Phase 5: Agentified Dynamically Collaborating Business Objects
Frequency of Change
1970
1975
1990
2002
???

Figure 2.1: Towards Loosely Coupled Software Systems

software tests must be adapted to which change then the automated tests can become unreliable. We cannot write or adapt an automated test if we don't know exactly which system elements must be checked for what. Hence, we often fail to keep track of requirements because we don't understand all the details needed to check a requirement. In this thesis, I present an approach that enables us to automatically detect violated or contradicting requirements even if the people who write down the requirements do not know in particular which system element will be checked for what.

2.2 The Running Example: A Privacy Policy

The following privacy policy is used as an example for crosscutting requirements throughout the thesis:

> All components handling personal data must be inaccessible to all components used in the workflow 'Create Report' because a report created in this workflow must not contain personal data.

How can the system be checked for compliance with this privacy policy? Which of the frequently changing elements of which frequently changing system artefact in which frequently changing contexts are affected by this requirement? For instance, which model element(s) of which UML diagram should be checked for what? How can we keep track of this requirement in loosely coupled, complex and quickly changing software systems in which no one understands all the details needed to check each requirement? The answer starts in the next section.

2.3 Objectives

This section describes the objectives of my approach. The next section will sketch the solutions I propose to meet these objectives.

Goal: Adaptive Approach

One crosscutting requirement affects 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. It is expensive to check all elements involved whether to adapt them each time the requirement changes. Moreover, it is expensive to check the system if the changed requirement newly applies to any other elements that were previously not involved in it. Even if the requirement itself does not change but any other system artefact elements shall be modified, it is expensive to check each new or modified element which other requirements apply to it. Furthermore, the system gets less comprehensible if the same crosscutting requirement is specified at every affected place. Redundancy causes information overload and can result in inconsistency. Hence, I propose that *one* requirement expression should address *all* of the elements involved, and this description should be **adaptive**: it should allow for automatic identification of all the system elements involved in the requirement. 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 privacy policy should be expressed in a way that enables software tools to automatically identify all constrained elements: the adaptive specification should enable software tools to identify those components that handle personal data and those components used in the workflow 'Create Report' automatically.

Goal: Independent of Artefact Types

In a system model, one crosscutting requirement affects several model elements that may not be associated with each other or may even belong to different models. In source code files, one crosscutting requirement is reflected in many different lines of code. Likewise, one crosscutting requirement can affect several places in configuration files. At runtime, one crosscutting requirement can affect several binary components that may not invoke each other or may even run on different platforms. In my opinion, it takes too much effort if the same crosscutting requirement must be stated newly for each software system artefact. Furthermore, the effort of writing down the requirement in the minutest artefact-specific details is unsuitable if these details are not important. I also want to avoid addressing all the artefact-specific details because they might reduce the comprehensibility of the requirement expression. Instead, I propose that a requirement expression should be independent of the modelling language, the programming language, or the middleware platform, in order to apply the same requirement expression to all these artefact types. Moreover, it doesn't have to be re-written if a different artefact type (or version) is used. For example, the privacy policy (see section 2.2) 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.

Goal: Detect Violated or Contradicting Requirements

Violated or contradicting requirements should be detected as soon as possible during the software development process. I propose a requirement specification technique that enable tools to find both contradicting and violated requirements automatically. In order to reach this goal, algorithms for detecting violated or contradicting requirements are needed. For example, these algorithms should detect any artefact element that violates the privacy policy, and they should detect which other requirement contradicts the privacy policy.

2.4 Structure of this Thesis

This section outlines the contribution of this thesis by sketching its chapters. The first chapter has explained the basic terms of components, UML, and requirements engineering. This second chapter has explained important software engineering problems in section 2.1, gave a short example in section 2.2, now outlines how I propose solving these problems in this thesis.

CoCons

In chapter 3, I will introduce a new requirements specification technique called context-based constraints (CoCons). Their basic idea can be explained in just a few sentences:

1. Metadata is defined as 'data about data' in [Pic00]. I suggest annotating the system artefact elements with formatted metadata called 'context properties'. A context property describes its element's context. As discussed in section 3.2, context is any information that can be used to characterize the situation of an element.
2. A CoCon is a constraint that expresses a condition on how system elements must (or must not) relate to each other. For instance, a CoCon can express the privacy policy in section 2.2 as the condition that certain elements *must be inaccessible to* certain other elements. It can select its constrained elements via their context properties: only those elements whose context property values fit the CoCon's 'context condition' must fulfil the constraint. Section 3.3 will explain this new concept in detail.

Applying CoCons

In chapter 4, I will discuss how to use CoCons after they have been written down. First, section 4.1 will explain why to consider requirements as invariants in system modifications: when adapting a system to new, altered, or deleted requirements, existing requirements should not unintentionally be violated. Afterwards, two algorithms for detecting violated or contradicting CoCons will be presented. Section 4.2 will explain how to detect if an artefact element does not comply with a CoCon's condition on how it must (or must not) relate to another artefact element. Afterwards, section 4.3 will examine how to detect if one CoCon contradicts another CoCon. Finally, section 4.5 will introduce concepts for maintaining context property values in system modifications because violated or contradicting CoCons cannot be detected at all if they refer to wrong, outdated, or missing context property values.

Which Requirements?

In chapter 5, I will suggest 22 different types of CoCons that define different requirements for component-based systems. They are grouped into five families:

- Access Permission CoCons express which components *must (or must not) be accessible to* which other components as discussed in section 5.2.
- Communication CoCons control whether a method call between components *must be (or must not be)* handled as examined in section 5.3.
- Distribution CoCons express which components *must (or must not) be allocated to* which computers as shown in section 5.4.
- Information-Need CoCons express which users *must (or must not) be notified of* which documents at runtime as described in section 5.5.

- Inter-Value CoCons express whether elements in a certain context must (or must not) reside in another context as explained in section 5.6.

Checking UML Models

In chapter 6, I will demonstrate how to check a UML model for compliance with the CoCon-predicates proposed in chapter 5. First, section 6.1 will explain how to integrate the CoCons of chapter 5 into the UML metamodel because the notion of CoCons is not part of the UML yet. Then, section 6.2 will compare CoCons with the UML's standard constraint language OCL. This comparison will reveal why CoCons are a new concept in detail.

Methodical Guidance

In chapter 7, I will propose a method for identifying and applying CoCons during requirements analysis. First, section 7.2 will describe how to discover contexts and business rules during requirements elicitation. Afterwards, section 7.3 will explain how to negotiate the CoCons discovered during requirements elicitation with all stakeholders in order to arrive at a set of agreed upon requirements. Next, section 7.4.will provide assistance in specifying the agreed upon CoCons formally. Finally, 7.5 suggests how to check the specification for omitted, wrong, ambiguous and inconsistent parts.

Conclusion

To conclude, I will summarize the limitations and benefits of the presented approach in chapter 8. But first, we start with its introduction in the next chapter.

3. Context-Based Constraints (CoCons)

3.1 Overview

Keep It Simple

Requirements engineering is not only a process of discovering and specifying requirements, it is also a process of facilitating effective communication of these requirements among different stakeholders. The way in which requirements are documented plays an important role in ensuring that they can be read, analysed, (re-)written, and validated. In order to *easily* communicate requirements among different stakeholders, I stick to the *simple* solutions throughout this thesis. Keeping it simple results in a limited expressiveness but also leads to benefits discussed in the oncoming sections.

Syntax & Semantics

This chapter will propose a new concept for expressing requirements. In textual notations, **syntax** is described by the set of characters used (alphabet) and their possible sequences. Syntactical issues purely focus on the notation, completely disregarding any intention (semantics) behind the notation. The **semantics** of a language tell us about the meaning of each construct of the language in question. This is usually done by explaining the constructs of the language in terms of already known (and hopefully well understood) concepts.

Degree of Formality

According to [Rum98], several degrees of formality of a notation exist. If the syntactic shape of a notation is precisely defined, then the syntax is formalised. However, based on the syntax the meaning of the notation has still to be defined. The benefits of a 'formalization' mapping are less that it is 'formal' afterwards, but more that the mapping of a new notation into a given formalism let properties of the new notation become apparent that had been hidden before. This allows a deeper and better understanding of the new notation.

Artefact-Type-Independent

Different artefact types are used at different abstraction levels throughout the software engineering process. For instance, UML models or other specification techniques can be used at the design level. This chapter discusses which artefact 'element' is involved in which requirements *independent* of specific artefact types. Chapter 6 will define the artefact-type-specific semantics for UML 2.0 models.

Structure of this Chapter

As sketched in section 2.4, a new constraint technique is presented here that adapts to changes more easily because it indirectly selects its constrained elements according to their metadata. This metadata must be formatted in order to use it for the constraint techniques. First, section 3.2 proposes a syntax (format) for metadata called 'context properties'. Then, the new constraint technique that refers to this formatted metadata is explained in section 3.3. Finally, section 3.4 discusses how to turn these constraints into business rules by considering events and actions. Each section starts with an *informal*, intuitive description of the presented concept, and then defines a *textual syntax* before providing *formal* semantics that explain the new concepts in terms of already known concepts.

3.2 Introducing Context Properties

3.2.1 What is Context?

As explained in section 2.4, this thesis presents a new notion of constraints that select their constrained elements according the element's metadata. Hypothetically, any kind of metadata of an element can be used to identify whether this element is a constrained element or not. However, I focus on only one kind of metadata here: I only consider metadata describing the *context* of elements. This section presents my definition of 'context' .

Linguistic Context

First, the **linguistic** notion of context defined in the dictionary of philosophy ([Ang92]) is briefly explained:

> **context** (L. *contexere*, 'to weave together.' from *con*, 'with', and *texere*, 'to weave'): The sum total of meanings (associations, ideas, assumptions, preconceptions, etc.) that (a) are intimately related to a thing, (b) provide the origins for, and (c) influence our attitudes, perspectives, judgments, and knowledge of that thing.

If something is seen in context or if it is put into context, it is considered with all the necessary factors that are related to it so that it can be properly understood rather than just being considered on its own. If something is taken out of context, it is only considered on its own, but the circumstances in which it was defined are ignored. In that case, it can mean something different from the meaning that was intended.

Avoiding Complexity

This thesis focuses on the context of software system elements. It uses context for one specific purpose explained in section 2.4: it needs context in order to distinguish those software system elements that reside in the same context from other elements that don't. However, each element resides in an infinite number of contexts – according to [SG89, KS96], 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 [Hir00]. Only limited context models can be handled. Thus, this section presents a simple and limited context model. But first, some related research is discussed.

Internal Context

The context models used in software engineering all focus on internal context of software systems as explained next. 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, the name of a component, the name of a method, or the name of a method's parameter are internal elements because these names are defined in the system's artefacts. On the contrary, an **external element** is not contained in any of the system's artefacts. An **internal context** of a software system element 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 [Szy97]. 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 ([OMG03b]), 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. In general, the context definitions used in software engineering only consider internal context.

External Context

I propose also to take non-internal contexts into account. Context should assist in selecting constrained elements regardless whether their context is part of the system or not. The system, however, must not necessarily be modified in order to manage additional context as internal context. Instead, the additional context can be managed in an external repository that refers to the system's elements in order to identify those system elements that reside in the external context. Hence, context can be taken into account even if it is not part of the system at all. As soon as an externally defined context is added to the system's artefacts it becomes internal context.

No Context of Context

The internal context of an artefact element is expressed via other artefact elements. Therefore, the **context of a context** can be expressed because the context itself is an artefact element whose context again can be described via another artefact element (whose context again can be described via another artefact element and so on). Considering the context of a context can result in very complex context models typically developed in artificial intelligence (AI) research (see section 3.2.6). I ignore such complex context models because I express context of an element to distinguish those elements that reside in the same context from other elements that don't. In order to identify whether an element directly resides in a context, it is not necessary to consider the context of this context. Only the direct context of an element is needed in order to distinguish it from the other elements that do not directly reside in this context.

Situational Context

Situational context is defined in [Dey01] as 'any information that can be used to characterize the situation of an entity'. This notion needs a precise definition of 'situation'. In situation calculus ([Dev91]), **situation** is defined as a snapshot of the world at a particular point in time. While situations are a complete state of the world at a certain time, our knowledge of a situation is necessarily incomplete. Hence, an agent can only pick out a structured part of the reality that represents a situation. This definition suits well for this thesis 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.

Context (Intuitive Def.)

I express **context** in order to identify those elements that are involved in a requirement and informally define it as follows:

- The context of a software system element characterizes the situation(s) in which the element resides.
- 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 next sections present a syntax for expressing context of software elements and provides some examples before defining the semantics formally.

3.2.2 Context Properties: Formatted Metadata Describing Elements

The context of an element can be expressed as metadata (called ‘meta-attributes’ in [SSR92, SSR94]). Over the last decade, metadata concepts are increasingly used to handle the required flexibility of global software and information infrastructures (see, e.g., [Bre94]). The explicit introduction of metadata concepts for many purposes results in a wide variety of specific definitions of metadata as discussed, e.g., in [Bus02]. This section proposes a metadata syntax (or ‘format’) for describing the context of elements. Without an agreed syntax for the metadata, tools cannot automatically decode it. Hence, this section *informally* defines a syntax and its semantics for context. The same syntax and semantics are *formally* defined afterwards in section 3.2.4.

Context Property

According to [Bak00], the attribute-value pair model is the commonly used format for defining metadata today. As well, this thesis suggests expressing context in the simple attribute-value syntax. As an alternative, the context of an element could be expressed in more complex data schemas. For instance, the context of an element can be expressed in hierarchical, relational, or object-oriented data schemas. Complex data schema types provide a better expressiveness, but are more difficult to understand. The rationale for using a non-hierarchic, non-relational, and non-object-oriented data scheme is simplicity. The concepts introduced in this thesis are based on the attribute-value syntax. Section 3.3.3 will discuss how to adopt this simple approach to more complex data schemas. But first, let’s keep it simple.

In this thesis, a **context property** is a *typed attribute*: it consists of a name and a set of values as explained next and illustrated in figure 3.1. First, the syntax of context properties is defined via BNF rules. Afterwards, this section informally explains the semantics of context properties.

BNF

The standard technique for defining the syntax of a language is the Backus-Naur Form (BNF), where “::=” stands for the definition, “`Text`” for a nonterminal symbol and “**TEXT**” for a terminal symbol. Square brackets surround [optional items], curly brackets surround {items that can repeat} zero or more times, and a vertical line ‘|’ separates alternatives.

Textual Syntax

When associating values of one context property with an element, the following syntax (or format) is used:

Syntax for Associating Context Property Values With an Element

DirectConPropValues	::=	ContextPropertyName [‘**(**’ ElementName‘**)**’] ‘**:**’ ContextPropertyValue {‘**,**’ ContextPropertyValue}

Context Property Name

A context property is a typed attribute that consists of a name and one or more values. The context property **name** (called `ContextPropertyName` in the syntax definition given above) groups semantically related contexts. For example, the context property name ‘Workflow’ groups the names of the workflows in which the associated element is used.

Context Property Value

The BNF rule `ContextPropertyValue` defines the valid values of one context property name. A subset of the valid values can be associated with a single element for each context property name. These context property **values** describe how or where this element is used – they describe the context (as discussed in section 3.2.1) of this element. The name of the context property stays the same when associating its values with several elements, while its values might vary for each element. If a context property value v is associated with a runtime artefact element e then it represents the **current context** of e at runtime.

Example The values of the context property named ‘Workflow’, e.g., reflect in which workflows the associated element is used as discussed in section 3.2.3. For instance, the four values allowed for Workflow can be `ContextPropertyValue := ‘New Contract’ | ‘Delete Contract’ | ‘Integrate Two Contracts’ | ‘Split One Contract’, ‘Create Report’`. They are called **valid values** because no other values of the context property Workflow can be associated with an element. The set of valid values defines the *type* of the context property - it determines which values are allowed. The expression “`Workflow(ContractManagement): ‘Delete Contract’, ‘Create Contract’, ‘Integrate Two Contracts’`” associates the three values ‘Delete Contract’, ‘Create Contract’ and ‘Integrate Two Contracts’ of the context property ‘Workflow’ with the element ‘ContractManagement’. Providing the name of the associated element ‘ContractManagement’ in round brackets after the context property name is not required if the values are associated with the element already as depicted via a dotted line in figure 3.1.

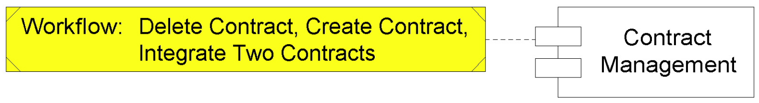
Workflow: Delete Contract, Create Contract, Integrate Two Contracts
Contract Management

Figure 3.1: Graphical Notation for Context Properties

Graphical Notation In figure 3.1, three of the valid values of the context property ‘Workflow’ are associated with the component ‘Contract Management’. They describe, in which workflows the component Contract Management is used. The context property symbol resembles the UML symbol for comments because both describe the model element to which they are attached. The context property symbol is assigned to one model element and contains the name and values of one context property associated with this model element.

3.2.3 Context Property Examples

This sections lists examples of *general* context properties for component-based systems. They are called general because they can be applied to any element type. More examples for context properties useful in component-based software systems are provided in [Büb02a].

‘Workflow’ The values of the context property ‘Workflow’ reflect the workflows in which the associated element is used. Hiding avoidable granularity by only considering *static aspects of behaviour* (= nothing but workflow names) enables developers to ignore details. Otherwise, the complexity would get out of hand. The goal of the approach presented here is to keep the requirement specifications as straightforward as possible. If preferred, the term ‘Business Process’ or ‘Use Case’ may be used instead of ‘Workflow’.

‘Operational Area’ The values of the context property ‘Operational Area’ describe, in which department(s), module(s), or domain(s) the associated element is used. It provides an organisational perspective.

‘Classified Data’ The values of the context property ‘Classified Data’ signal whether an element handles confidential data. In many examples throughout this thesis a refined version of this context property is used: the values 'True' or 'False' of the context property '**Personal Data**' signal whether an element handles data of private nature or not.

...are External Context

Such context information is typically not part of a system's source code. 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.

Quick Tour $\xrightarrow{goto}$ Section 3.3

The primary benefit of enriching elements with context properties is revealed in section 3.3, where they are used to specify 'context-based constraints'. Readers in a hurry can skip their formal definition of context properties in the next sections and proceed in section 3.3 on page 30.

3.2.4 Formal Definition of Context Properties

The previous section has *informally* defined a context property as a *typed attribute*. A formal definition of these typed attributes will be needed in section 3.3.7. However, typed attributes can be formalized in many ways. This section first defines a formal syntax and then the formal semantics of context properties in order to provide a basis for section 3.3.7. Section 3.3.3 will discuss how to replace the formal definition of context properties presented here with other context models

Def. Context Property Syntax

The formal syntax definition of a context property as 2-tupel (cp, VV^{cp}) is introduced in [Büb00a] and refined here:

1. CP is the set of the names of all context properties used in the system.
2. $cp \in CP$ is the **name** of one context property (e.g. cp_1 ='*workflow*')
3. VV^{cp} is the set of **valid values** for one context property $cp \in CP$. For instance, the five values allowed for cp_1 can be $VV^{cp_1} = \{$ 'New Contract', 'Delete Contract', 'Integrate Two Contracts', 'Split One Contract', 'Create Report'$\}$. They are called valid values because only values that are contained in VV^{cp} can be associated with an element.

In section 3.2.2, a textual syntax for context properties was defined. This textual syntax corresponds to the formal syntax defined here as follows: the BNF rule `ContextPropertyName` defines the set CP of the names of all context properties used in the system, and the BNF rule `ContextPropertyValue` defines the set of valid values VV^{cp}. The formal syntax defined here is more precise than the textual syntax defined in the previous section because VV^{cp} contains the valid values of *one* context property cp, while `ContextPropertyValue` can contain the valid values of *all* context properties $\forall i : cp_i \in CP$.

Def. Context Property Semantics

A context property value can be associated with an element in order to describe the element's context. This can be formally defined as follows:

1. E is the set of all elements in the system (model), e.g. e_1 = the component '*ContractManagement*'.
2. Multiple values $v_{1...n} \in VV^{cp}$ can be *directly associated* with one element $e \in E$ via the **directvalues** mapping:

$$directvalues_{cp} : E \rightarrow \mathcal{P}^{VV^{cp}}$$

For one context property $cp \in CP$ it maps an element $e \in E$ to a subset of *cp*'s valid Values VV^{cp} – denoted as an element in the power set $\mathcal{P}^{VV^{cp}}$ (e.g. $directvalues_{cp_1}(e_1)$ = {'Integrate Two Contracts'}).

3. Optionally, '**inter-value constraints**' can be defined for a context property *cp*. An inter-value constraint forbids or enforces certain values to be contained in $directvalues_{cp}(e)$ according to other values in $directvalues_{cp}(e)$.

The textual syntax for associating context property values with an element is defined in section 3.2.2. The BNF rule `ElementName` defines the set E of all elements in the system (model). The subset of the valid values VV^{cp} of the context property *cp* = 'Workflow' can be associated with the element e either via the formal syntax presented here or via the textual syntax defined in section 3.2.2. The BNF rule `DirectConPropValues` represents the mapping $directvalues_{cp}(e)$, where e corresponds to `Elementname`, and *cp* to `ContextPropertyName`. For example, the following two notations have the same semantics:

- $directvalues_{Workflow}(ContractManagement)$ = { '*Integrate Two Contracts*', '*Split One Contract*'}.
- `Workflow(ContractManagement): 'Integrate Two Contracts', 'Split One Contract'`

Inter-Value Constraints

Defining the valid values does not prevent contradicting values. For instance, the valid values of the context property 'Personal Data' can be defined as {'True', 'False'}. However, this definition does not prevent associating both 'True' and 'False' with the same element. Inter-Value constraints can specify dependencies between the context property values of an element. They can express that if an element is associated with a certain context property value then it must (or must not) be associated with another context property value. An inter-value constraint can, e.g., state that an element having the value 'True' must not be associated with 'False' and vice versa. Moreover, inter-value constraint express more than contradicting values. They can express whether certain values of one element can be derived from other values associated with the same element. If, e.g., the workflow 'Integrate Two Contracts' is part of the workflow 'CustomerMarriage' then the value 'CustomerMarriage' should be associated with all elements which already have the value 'Integrate Two Contracts' for 'Workflow'. An inter-value constraint can express this dependency by stating that an element having the value 'Integrate Two Contracts' must also have the value 'Customer Marriage'. Inter-Value CoCons introduced in section 5.6 can define such inter-value constraints that express dependencies between context property values.

3.2.5 Useful Context Property (Stereo-)Types

No Context Classification

As discussed in section 3.2.1, it is not possible to list the complete context of an element. For example, a physical context a copy of this thesis lying in the office can be defined as 'is in the office'. This is not the only physical context of the document, though. Instead, endless physical contexts exist: the document also resides in the physical context 'in the house containing the office', or 'on a little blue green planet orbiting a small unregarded yellow sun at a distance of roughly ninety-two million miles far out in the uncharted backwaters of the unfashionable end of the western spiral arm of the Galaxy'([Ada79]) — an element resides in numberless physical contexts.

Physical context is not the only kind of context. Countless other kinds of context exist, e.g. conceptual, social, historical, or cultural context, because for each of the context of an element a new kind of context could be defined. Hence, this thesis neither can list all contexts of an element nor can it list all kinds of contexts. Some examples of typical contexts for component-based systems are provided in [Büb02a]. Even these examples are not a complete list of all contexts that can be considered in component-based systems. Instead of providing an incomplete list of context properties, section 7.2 will discuss methodical guidance on identifying relevant context properties for a specific application domain.

(Stereo-) Types

A classification of contexts cannot be complete. Nevertheless, this section proposes to distinguish three types of context properties according to a technical criterion on their valid values: a **context property type** groups those context properties whose values can automatically be handled by a tool in a special way. Context property types are defined via UML stereotypes here. According to [OMG03b], a stereotype refines an already existing concept. Stereotypes may extend the semantics, but not the structure of pre-existing concepts. Two useful stereotypes for refining the semantics of context properties are suggested:

≪Number≫ : The valid values of a ≪Number≫ property are numbers. In addition to normal context properties, the particular measurement is added in round brackets to the context property name, e.g. '≪Number≫ Amount (in Instances)' . Context properties whose values are number can be used by tools for calculation.

≪System≫ : The current values of a ≪System≫ property can be extracted from the system each time when they are retrieved. For instance, the current local time of an element is a ≪System≫ property.

These two context property types can overlap: a ≪Number≫ property can also be a ≪System≫ property. For instance, the 'current local time' of an element used as an example for ≪System≫ properties above is also a ≪Number≫ property.

The third type of context properties are the normal context properties who neither have numbers as values nor have values that can be queried from the system. They are not refined via a stereotype because they do not refine the context property definition given in this chapter.

3.2.6 Research Related To Context Properties

Formalizations of Context

The problem of context has a long tradition in different areas of artificial intelligence (AI). The issue of formalizing context has become widely discussed in the late 80s, when McCarthy argued that formalizing context is a crucial step towards the solution of the following problem: 'Whenever we write an axiom, a critic can say that the axiom is true only in a certain context. With a little ingenuity the critic can usually devise a more general context in which the precise form of the axiom doesn't hold' ([McC87]). Since McCarthy, two main formalisations have been proposed in AI: *propositional logic of contexts* (PLC), and *Local Models Semantics / Multicontext Systems* (LMS/MCS). An in depth comparison of both approaches is provided in [BS02], and a survey on these and other approaches on formalizing context is given in [AS96]. However, none of these formalizations are discussed in detail here because they all express context of *propositional logic formulas*. The discussion on considering context in logical formulas continues in section 3.3.6.

Semantic Values

As summarized in [SSR94], a concept similar to context properties was discussed in the 90ties: database objects are annotated via intensional description called 'semantic values' ([SG89, SSR92]) 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 heterogeneity among them. Still, the concepts are similar because they both can denote elements residing in the same context.

Domains

Likewise, 'domains' ([ST94]) are similar to context properties. Domains are typically used in policy management. In contrast to context properties, a domain consists of a single name and not of a name and value(s). Moreover, domains are hierarchical.

Tagged Values

Many notations for writing down metadata exist and can be used for expressing the context of elements. In UML, tagged values can be used to express context properties. However, in version 1.4 of UML or later, the 'tag definition' of a tagged value must be associated with a stereotype. In contrast, a context property definition must not necessarily be associated with a stereotype.

Stereotypes or Packages...

A context property groups model elements that share a context. Object-oriented grouping mechanisms like *inheritance*, *stereotypes* ([BGJ99]) or *packages* are not used because the values of a context property associated with one artefact element might vary in different configurations or even change at runtime. Multiple values of several context properties can be associated with the same element e. Let a system have n different valid values in $VV^{Workflow}$. Then for each $v_i \in VV^{Workflow}$ an element e 'is' or 'is not' (= 2 possibilities) associated with v_i. Allowing for all possible combinations, you would need up to 2^n different stereotypes in UML 1.3 because according to [OMG99], it is not allowed to have more than one stereotype per element.

...Cannot Change at Runtime

This multiplicity has changed to 0..$*$ in UML 1.4 and later. So that in UML 1.4 only n different stereotypes are needed for associating any of n valid values of the context property 'Workflow' with it. However, even UML 1.4 multi-inheritance of stereotypes is unsuitable, since usually the values of more than one context property are associated with one model element. Considering only one additional context property, e.g. 'Operational Area' with m valid values that *overlaps* with 'Workflow' having n valid values would result in up to $n \times m$ stereotypes in UML 1.4, and up to 2^{n+m} in UML 1.3. Furthermore, the values of a context property might vary in different configurations or change at runtime. A model element is not supposed to change its stereotype or its package at runtime. One context property can be associated with different types of model elements. For example, the values of 'Workflow' can be associated both with 'classes' in a class diagram and with 'components' in a component diagram. Using packages or inheritance is not as flexible. According to [OMG03b], stereotypes can group model elements of different types via the baseClass attribute, too. However, this 'feature' has to be used carefully. Moreover, an element is not supposed to change its stereotype, its inheritance, 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, artefact element types, or platforms.

Metadata in Java

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 new 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.

The primary benefit of enriching elements with context properties is revealed in the next section, where elements are selected according to their context property value in order to identify those elements that are involved in a requirement.

3.3 Introducing Context-Based Constraints (CoCons)

This section presents a new constraint technique called 'CoCons' for specifying crosscutting requirements. It was introduced in [Büb02b] and refined in [BB05]. First, CoCons are *informally* explained in section 3.3.1. Then, section 3.3.2 and 3.3.3 define a textual syntax for expressing CoCons. As explained in section 2.3, a key goal of this thesis is checking system artefacts for compliance with CoCons automatically. Section 3.3.5 discusses how to achieve this goal by precisely defining the CoCon semantics. Afterwards, the *artefact-type-independent* CoCon semantics are formally defined in section 3.3.6 and 3.3.7. Finally, section 3.3.9 discusses related research.

3.3.1 Intuitive Definition of Context-Based Constraints

Intuitive Def. CoCon

A **context-based constraint** (CoCon) is 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). Another CoCon-predicate could express that certain elements must (or must not) be allocated to certain computers (distribution requirement). More CoCon-predicates will be introduced in chapter 5. The metamodel in figure 3.2 shows the abstract syntax for CoCons. Its metaclasses are informally explained next.

Example A

If values of the context property 'Personal Data' (see section 3.2.3) are associated with the system components then a CoCon can state that

"*All components whose context property 'Personal Data' has the value 'True' must be inaccessible to the component 'WebServer'*" **(Example A)**.

This constraint is based on the context 'Personal Data' of the components – it is a *context-based* constraint that expresses a CoCon-predicate on how its constrained elements must relate to each other: they must not access each other.

Example B

Another example requirement is based on the context 'Workflow' of the components involved:

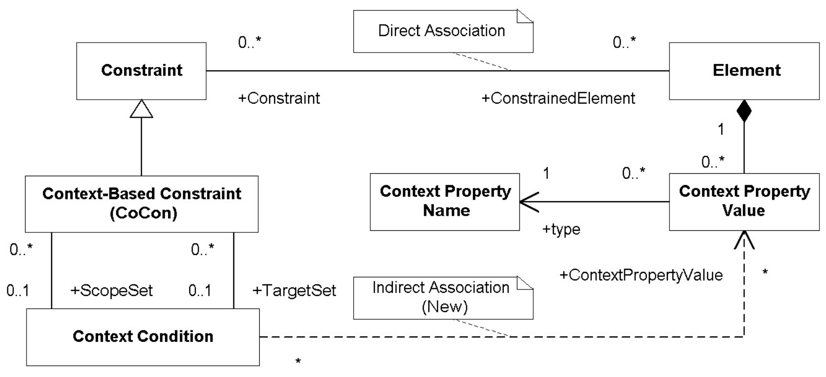

Figure 3.2: The CoCon Metamodel

"*The component 'EmployeeManagement' must be inaccessible to all components whose context property 'Workflow' contains the value 'Create Report'*" **(Example B)**.

Context Condition

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. As discussed later on in section 3.3.8, context conditions are similar to point cuts in aspect-oriented programming. It may be restricted to artefact elements of one element type. In UML, a model 'element type' is called 'metaclass'. As result of a **range**, no (model) elements of other types are selected even if their context property values fit the context condition. Throughout this thesis, the context conditions are mostly restricted to 'components'.

Two Sets

A CoCon can relate many *sets* of constrained elements. I only discuss CoCons that relate *two* sets of elements. Figure 3.3 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.

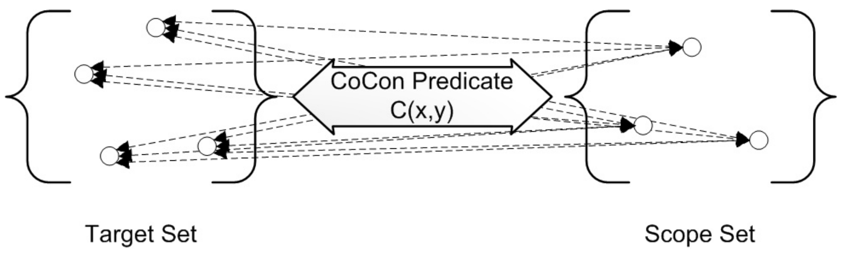

Figure 3.3: A CoCon Relates any Element of the 'Target Set' with any Element of the 'Scope Set'

'Target Set' and 'Scope Set'

The two sets related by a CoCon are called 'target set' and 'scope set'. In example A, the **'target set'** is selected via the following context condition: "*All components whose context property 'Personal Data' has the value 'True'*". And in example A, the **'scope set'** of the CoCon contains a single element – the component 'WebServer'. Both sets can contain any number of elements, as illustrated in example B. Yet, in some cases the names 'target set' and 'scope set' do not seem appropriate. Mixing example A and example B, a CoCon could state that "*All components whose context property 'Personal Data' has the value 'True'* (Set_1) *must be inaccessible to all components whose context property 'Workflow' contains the value 'Create Report ' (Set_2)*". Set_2 is called the scope set here. Nevertheless, which part of the system is the scope in this example? Should those elements of the system be called 'scope' which are inaccessible (Set_1), or does 'scope' refer to all elements (in Set_2) that cannot access the elements in Set_1? Should Set_1 be called 'scope set' and Set_2 'target set' or vice versa? Unfortunately, there is no intuitive answer in all cases. Nevertheless, it is better to give each set a name instead of calling it Set_1 or Set_2. For most CoCon-predicates (see section 5) , the names 'target set' for Set_1 and 'scope set' for Set_2 fit well. Perhaps future research reveals names for these sets that always fit well.

Selecting Set Elements...

Both target set elements and scope set elements of a CoCon can be either directly or indirectly selected:

...directly : A set element can be **directly** associated with the CoCon. In example A, the CoCon is *directly* associated with the 'WebServer' component by naming this component. This unambiguously identifies this component to be element of this scope set.

...indirectly (new) : **Indirect selection** is the key concept of context-based constraints. Set elements can *indirectly* be associated with a CoCon via a context condition. The scope set in example B contains all the components whose context property 'Workflow' contains the value 'Create Report'. These scope set elements are anonymous. They are neither directly named nor directly associated, but described indirectly according to their context property values. If no element fulfils the context condition, the set is empty. This simply means that the CoCon does not apply to any element at all.

This indirect selection is represented as a dotted line in fig. 3.2 because it defines a dependency that only exists if the element's context property values fit the context condition. The indirectly selected elements are identified by evaluating the context condition each time when an system artefact is checked for whether it complies with the CoCon.

Example A+B = Privacy Policy

The privacy policy explained in section 2.2 can informally be expressed via a CoCon as follows:

> *All components handling personal data must be inaccessible to all components used in the workflow 'create report'.*

This expression in natural language is a combination of example A and example B. The target set elements are described as *all components handling personal data*, the scope set elements are *all components used in the workflow 'create report'*, and the CoCon-predicate is *x must be inaccessible to y*, where x represents the scope set elements and y the target set elements. The example above is expressed in natural language – the next sections will define a textual syntax for CoCons that enables both computers and business experts to understand the expressed requirement.

Quick Tour $\xrightarrow{goto}$ Chapter 4

However, why should anyone use a textual syntax and write down CoCons at all? Answers to this question will be given in chapter 4. It will discuss algorithms that detect violated or contradicting requirements automatically if the requirement has been written down as a CoCon. Quick readers can skip the syntax details and proceed with chapter 4 on page

53. In BASIC: `GOTO 4`. Nevertheless, be careful – according to [Dij68], jump instructions are “considered harmful”. Even if you plan to skip the syntax details, you might take a look at the formal CoCon semantics explained in sections 3.3.5 and 3.3.6 before proceeding with chapter 4.

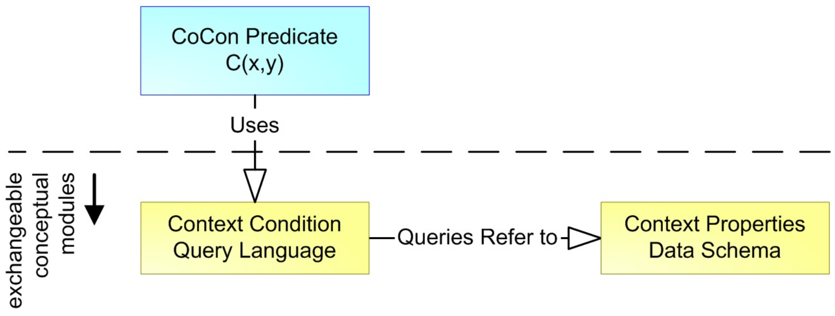
CoCon Predicate
C(x,y)
Uses
exchangeable conceptual modules
Context Condition Query Language
Queries Refer to
Context Properties Data Schema

Figure 3.4: The Replaceable Conceptual Modules of CoCons

Replaceable Modules of CoCons

Section 3.3.3 will discuss different query languages for expressing context conditions. The context condition query language depends on the context property data schema used. The semantics of a *CoCon* stay the same even if the *context property data schema* and the *context condition query language* are exchanged. Figure 3.4 illustrates these **replaceable conceptual modules** of a CoCon.

Additionally, a **CoCon attribute** can define details of its CoCon. Each attribute has a name and one or more value(s).

CoConAUTHOR

A CoCon’s author can be defined via the attribute **CoConAUTHOR** .

COMMENT

A comment describing the CoCon can be defined via the attribute `COMMENT`.

CoConNAME

A CoCon can be named via the attribute `CoConNAME`. This name must be unique because it is used to refer to this CoCon.

PRIORITY

A CoCon defines relation between its scope set elements and its target set elements. It relates each element of one set to each element of the other set and defines a CoCon-predicate between each pair of related elements. However, CoCons can contradict each other if they refer to the same element(s) but state contradicting CoCon-predicates for these elements. If the contradicting CoCons have different priority`PRIORITY` then only the CoCon with the highest priority applies. If this CoCon is invalid because its scope or target set is empty then the next CoCon with the second-highest priority applies.

3.3.2 The Common CoCon Syntax

This section introduces the common textual syntax of CoCon. It is called ‘common’ syntax because it defines the basic CoCon syntax for *all* CoCon-predicates. Certain BNF rules differ between different CoCon-predicates. They must be refined for each CoCon-predicate as explained below. The semantics of the terms used here have informally been explained in section 3.3.1, and will formally be defined in section 3.3.6.

`Rule`$+^{(OR\ |\ AND)}$

BNF Rules concerning the separators ‘,’, ‘OR’ or ‘AND’ are abbreviated: “`Rule { Separator Rule }*`” is abbreviated “`(Rule)`$+^{Separator}$”.

Common Syntax of Context-Based Constraints

`CoCon`	::=	`TargetSet` ‘**MUST**’ `[PredicateOperation]` ‘**BE**’ `CoConPredicate ScopeSet` [‘**WITH**’ `(Attribute)`$+^{AND}$] [‘;’]
`PredicateOperation`	::=	‘**NOT**’ | ‘**ONLY**’
`TargetSet`	::=	`ElementSelection` [‘**AMONG WHOM**’ [‘**NOT**’] ‘**EXIST(S)**’ `ElementSelection`]
`ScopeSet`	::=	`ElementSelection` [‘**AMONG WHOM**’ [‘**NOT**’] ‘**EXIST(S)**’ `ElementSelection`]
`ElementSelection`	::=	`(SelectionExpr` | `(‘(’ElementSelection ‘)’)` `)`$+^{(OR\ \mid\ AND)}$
`SelectionExpr`	::=	`(IndirectSelection` | `DirectSelection)`$+^{(OR\ \mid\ AND)}$
`DirectSelection`	::=	`(`‘**THE**’ `[ElementTypeAlias] ElementType ElementName)` | ‘**THIS**’
`IndirectSelection`	::=	`Cardinality [ElementTypeAlias] ElementTypes [ContextCondition]`
`Cardinality`	::=	‘**ALL**’ | `Number` | `(`‘**BETWEEN**’ `LowerBoundNumber` ‘**AND**’ `UpperBoundNumber` ‘**OF ALL**’`)`
`Attribute`	::=	`AttributeName` ‘=’ `(AttributeValue)`$+^{Comma}$
`AttributeName`	::=	‘**COCONNAME**’ | ‘**COCONAUTHOR**’ | ‘**COMMENT**’ | ‘**PRIORITY**’

Privacy Policy Example

According to common BNF rules given above, the privacy policy explained in section 2.2 can be expressed as follows:

> *All components handling personal data* `MUST NOT BE ACCESSIBLE TO` *all components used in the workflow ‘create report’.*

Some parts of this statement are printed in *italics*, while other parts are printed in `UPPERCASE TYPE WRITER` font because this common CoCon syntax definition provided above is incomplete. Only the parts in `UPPERCASE TYPE WRITER` font are defined above. The *italic* parts of the example expression are not defined yet. In detail, the following rules are not defined here yet:

- The values of the `CoConPredicate` rule define the names of the CoCon-predicates. Details on defining a CoCon-predicate will be discussed in section 3.3.5.
- The `ContextCondition` rule defines a query language for indirectly selecting the elements constrained by the CoCon according to their context property values. Section 3.3.3 will suggest a query language and compare it with other query languages.
- The values of the `ElementType` and the `ElementTypes` rules define the *type* of the elements that are selected, e.g. components. They are not defined here yet because these values depend on the application domain of CoCons. ElementTypes suitable for component-based systems are defined in appendix A.

- A CoCon's target set or scope set can be restricted to elements of one type. If both sets are restricted to the same element type, e.g. components, then the `ElementTypeAlias` rule can be used to define an alias for one set, e.g. 'ScopeComponents', in order to distinguish the elements on one set from the elements in the other set. Section 3.3.4 discusses the semantics of `ElementTypeAlias` in more detail.
- The values of the `ElementName` rule refer to the names of individual elements of the system to which the CoCon is applied.
- The rule `PredicateOperation` allows to place the operation `NOT` or the operation `ONLY` after the keyword `MUST`. This **CoCon-predicate Operation** changes the semantics of a CoCon-predicate. Its effect is formally defined in section 3.3.6.
- The BNF rule `ElementTypes` names the *type* of the elements that are selected via this context condition, e.g. components.
- The terminal symbol '`THIS`' in the `DirectSelection` rule has the same meaning as '`self`' in OCL (see [CKM$^+$99b, WK99]): if the CoCon is directly associated with a model element then '`THIS`' *directly* refers to the associated model element. For example, if a CoCon is associated with a component then `THIS` refers to this component.
- The terminal symbol '`ELEMENTS`' in the `Restriction` rules is needed to specify *unrestricted* indirect selection of elements. It selects elements regardless of their metaclass. On the contrary, '`ALL COMPONENTS`' is a restricted selection. An example is given in section 5.6.4.
- The values of `ElementTypeAlias`, of `ElementName`, and of `AttributeValue` can be given in quotation marks. Either simple 'quotation marks' or double "quotation marks" are allowed.

Different alternatives for defining the `ContextCondition` rule are discussed in the next section.

3.3.3 The Context Property Query Language CPQL

The key concept of CoCon is the *indirect* selection of elements *according to their context*. The context condition for indirectly selecting the elements involved can be expressed in different query languages. First, this section sketches existing query languages. Next, it informally explains the minimal functionality a query language should provide in order to express context conditions. Finally, a syntax of a simple context property query language is suggested. The formal semantics of this query language will be defined in section 3.3.7.

Why a New Query Language?

A simple and flat attribute-value schema for context properties has been introduced in section 3.2. Of course, more complex data schemata for storing the context properties of one element, e.g. hierarchical, relational, or object-oriented schemata, can be used. 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. When applying CoCons, use your favourite query language. In terms of aspect-oriented programming (see section hui), a context condition is a point cut which determines the join points where to weave in the crosscutting advince. Hence, CPQL is a point cut definition language.

CPQL In this thesis, I stick to the non-hierarchical, non-relational, non-object-oriented data schema for context properties defined in section 3.2. The simple **c**ontext **p**roperty **q**uery **l**anguage **CPQL** defined next expresses context conditions for these non-hierarchical, non-relational, non-object-oriented context properties.

Query Capabilities Before introducing the CPQL syntax, details of context conditions are informally discussed. A query language used for expressing a context condition should provide the basic query capabilities discussed next. Of course, the question, which query capabilities a context condition should provide, depends on the personal taste and the application domain. As with the context property syntax, I suggest sticking to simple solutions. Maybe future research identifies reasons for reducing or enhancing the query capabilities proposed next.

Context Conditions refer... Two different kinds of context conditions exist. A context condition should be able to refer either to a set of context property values or to a single context property value.

...to a Set On the one hand, a context condition should be able to compare sets of context property values. According to section 3.2.4, one element e can be associated with a *set* of values for one context property cp via $values_{cp}(e)$. This set of context property values can be compared to another set of context property values via the following conditions:

- The condition 'CONTAINS' demands that one set is subset of or equal to ($\subseteq$) the other set. Moreover, other logical set conditions exist: `DOES NOT CONTAIN` ($\not\subseteq$), '=' (equals), `DOES NOT EQUAL` ($\neq$), `INTERSECTS WITH` (the-one-set $\cap$ the-other-set $\neq \emptyset$) and `DOES NOT INTERSECT WITH` (the-one-set $\cap$ the-other-set $= \emptyset$).
- Furthermore, two sets can be compared with logical conditions for comparing numbers: if the set A is compared via $>$, $\geq$, $<$, or $\leq$ to the set B then each value of A must fulfil this condition for each value of B. However, these comparison conditions are mostly used for comparing single values instead of sets as explained next.

...to a Value On the other hand, a context condition should be able to compare one context property value $v_{compare}$ with each value $v_i \in values_{cp}(e)$. If v_i is a string, than = (equals) and != (does not equal) are suitable conditions for comparing it with $v_{compare}$. If v_i is a number then — besides = and != — the usual logical conditions for comparing numbers should be provided by the query language: $>$, $\geq$, $<$, or $\leq$. If the set $values_{cp}(e)$ is compared with one value $v_{compare}$ then *each* $v_i \in values_{cp}(e)$ must be compared with $v_{compare}$. The context condition only selects e if all v_i fulfil the logical condition.

Cardinality A **cardinality** can limit the number of elements that are selected by a context condition. For example, a context condition can be limited to apply to only '3-7' elements. A CoCon having a limited cardinality is called a **flexible** CoCon because its constrained elements can freely be chosen from those elements that fulfil the context condition if more elements fulfil the context condition than specified in the range. An example will be discussed in section 5.4.4.

Total Selection One kind of indirect selection should exist that doesn't refer to context property values: the **total selection** simply selects *all* elements regardless of their context property values. However, it can be *restricted* to select only all elements of a certain metaclass, e.g. all 'components'. The `ContextCondition` part in the `Indirect Selection` rule of the CoCon syntax in section 3.3.2 is *optional* because total selections are defined by omitting a context condition: An example is given in section 5.5.4.

Combining Element Selections

In order to describe the elements of *one* set, direct and indirect **element selections** can be combined. For instance, a target set can contain "the component 'CustomerManagement' *or* all components that are used in the 'Create Report' workflow". In this example, a direct selection of the component 'CustomerManagement' is combined with an indirect selection via 'or' . As a result, this example will always select the component 'CustomerManagement' and additionally more components if they are used in the 'Create Report' workflow. Instead of the condition 'or' , the condition *and* can be used to combine two element selections, too. In this case, an element is only contained in the set if it fulfils both element selections combined via 'and' .

CPQL Syntax

The query capabilities of context conditions discussed above are considered in the following CPQL syntax definition:

The CPQL Syntax (Context Property Query Language)

`ContextCondition`	::=	**'WHERE'** `Query`+$^{(AND\ \mid\ OR)}$
`Query`	::=	`CompareTwoSets` | `SingleSetCondition`
`SingleSetCondition`	::=	`ContextPropertyName` (**'IS EMPTY'** | **'IS NOT EMPTY'**)
`CompareTwoSets`	::=	`ContextPropertyName` `CompareCondition` `SetOfConPropValues`
`SetOfConPropValues`	::=	`ContextPropertyValue` | (**'BOTH'** `ContextPropertyValue` (**'AND'** `ContextPropertyValue`)+) | (**'EITHER'** `ContextPropertyValue` (**'OR'** `ContextPropertyValue`)+ | (**'THE VALUES OF'** `ContextPropertyName`)
`CompareCondition`	::=	**'CONTAINS'** | **'DOES NOT CONTAIN'** | **'='** | **'DOES NOT EQUAL'** **'INTERSECTS WITH'** | **'DOES NOT INTERSECT WITH'** | **'<'** | **'>'** | **'<='** | **'>='**

The values of `ContextPropertyValue` and `ContextPropertyName` must be given in quotation marks. Either simple 'quotation marks' or double "quotation marks" are allowed.

Privacy Policy Example

According to the common BNF rules given in section 3.3.2, the privacy policy of section 2.2 can be specified as follows:

ALL COMPONENTS WHERE 'Personal Data' = 'True'
`MUST NOT BE ACCESSIBLE TO` *ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Create Report'* .

The two text parts in *ITALICS* represent the two CPQL expressions - one defines the target set elements, and the other one the scope set elements. In section 3.3.1, the same example has been expressed. But, in section 3.3.1 both context conditions for selecting the target set elements and the scope set elements were expressed in natural language. Now they are expressed in CPQL. Thus, the syntax for expressing CoCons is complete now. But, for which purpose do we need the syntax defined here? These questions will be addressed in chapter 4. It will discuss how to automatically detect requirements violations if the requirement has been written down as a CoCon.

Before discussing the formal CoCon semantics in section 3.3.5, the next section suggests adding path expressions the CPQL syntax.

In this section, basic query capabilities of query languages for expressing context conditions have been discussed. One query capability that turned out to be useful in the case studies, however, has not been addressed yet. It is suggested in the next section.

3.3.4 Navigation via Dot-Path-Notation

Navigation

When evaluating a context condition, all system (model) elements are checked for whether their context property values fit the context condition. In order to check all elements, step-by-step the context condition is checked for each single element. When checking a single element this element becomes the element **in focus**. The discussion of query capabilities of context condition in the previous section only considers conditions of the context property values of *one* element – the element in focus. In addition, this section suggests considering the context property values of *other* element when checking the element in focus, too.

Syntax

In order to refer to context property values of other elements in a context condition, the *path* to the other element must be given in order to navigate from the focused element to the other element. As in the object query language OQL or in the object constraint language in OCL ([WK99]), the **Dot-Path-notation** is suggested for expressing navigation paths here. It prevents ambiguity when addressing context property values.

The Syntax of the Dot-Path-Notation

ContextPropertyName ::=	[(ElementRole | Elementtype | ElementTypeAlias) {'**.**' (Elementtype | ElementTypeAlias) }*] ContextPropertyName

This syntax definition of the Dot-Path notation can be integrated into the query language syntax (see the previous section) because it refines the `ContextPropertyName` rule used in the CPQL syntax definition.

Semantics

If the other element is associated with the focussed element via (may be nested) associations then it is possible to navigate to this context property value along these associations via the dot-notation: the `ElementRole` of the element to which the path navigates is added as a prefix. Navigating via the dot-notation from one element to another is thoroughly explained in [WK99]. The same syntax and semantics as in OCL is used here for navigation along (may be several nested) associations. The result of navigating from the focussed element to another element when referring to a context property is that the context property values associated with the other element can be considered in a context condition as discussed in section 3.3.3.

Example

For instance, the element Product is associated with the element Contract, and the context property value 'Customer Marriage' of the context property 'Workflow' is associated with the element Contract. If Product is the currently focussed element then 'Contract.Workflow' refers to the value 'Customer Marriage' associated with Contract. A more detailed example is given in section 5.5.4.

Referring to the other Related Element

Besides navigating along associations, another notion of navigation is proposed next. A CoCon relates each element of its scope set with each element of its target set. A context condition on elements of one of these sets should be able to navigate to the other, related element in order to refer to the context property values of the other, related element. The other element's type (expressed via `Elementtype`) can be used to refer to the other element's context property values by adding the other element's *type* (followed by a dot) to the name of the context property as a prefix. If the other element has the same type as the focused element then an alias for referring to the other element (expressed via `ElementTypeAlias`) can be used to refer to the other element's context property values. The result of a writing down the name of a type as a prefix before the name of a context property is that only those values of this context property associated with instances of this 'other' element are taken into account when evaluation the context condition.

Example For instance, the context property 'Location' that describe the position of elements can be addressed in a context condition. It the target set contains elements of the type 'User' and the scope set contains elements of the type 'Component' then the Dot-Path notation facilitates stating clearly if an user's location or a component's location is addressed via 'User.Location' or 'Component.Location'. Again, are more detailed example is given in section 5.5.4. After informally introducing CoCons and CPQL and defining their textual syntax in the previous sections, the next sections define their formal semantics.

3.3.5 Two-Step Approach for Defining CoCon-Predicate Semantics

As explained in section 2.3, a key goal of this thesis is checking system artefacts for compliance with CoCons automatically. This section discusses *how to* formally define the semantics of a CoCon-predicate in order to achieve this goal. Section 3.3.6 will formally define the CoCon-predicate semantics as discussed here.

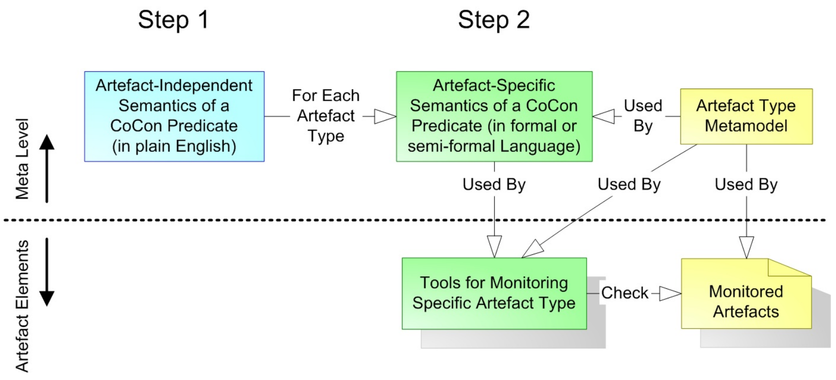
Step 1
Step 2
Artefact-Independent Semantics of a CoCon Predicate (in plain English)
For Each Artefact Type
Artefact-Specific Semantics of a CoCon Predicate (in formal or semi-formal Language)
Used By
Artefact Type Metamodel
Used By
Used By
Used By
Meta Level
Artefact Elements
Tools for Monitoring Specific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

Figure 3.5: Two-Step Approach for Defining the Semantics of CoCon-Predicates

Two-Step Approach 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 run-time. Figure 3.5 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 the `ACCESSIBLE`

`TO` CoCon-predicate used in the privacy policy example of section 2.2 are defined as follows: An `ACCESSIBLE TO` CoCon defines that its target set elements are accessible to its scope set elements. Chapter 5 lists more artefact-type-independent semantics definitions in plain English. Nevertheless, what exactly is the meaning of 'x must be accessible to y'? Does it, e.g., mean that certain relationships or dependencies must or must not exist in a UML diagram in order to comply with this CoCon? This is not defined here because it depends on the artefact type. Instead, it is defined in the artefact-type-specific semantics definition described below.

- 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. If a metamodel exists for the artefact type then the artefact-type-specific semantics are defined in terms and constructs of this metamodel in a formal or semi-formal language. For example, what is the meaning of 'x must be accessible to y' in UML 2.0 models? The artefact-type-specific semantics definition for `ACCESSIBLE TO` CoCons for UML 2.0 models is explained in chapter 6 and specified in appendix B starting on page 121 – the artefact-type-specific semantics definition is much longer (11 pages) then the artefact-type-independent semantics definition (in plain English: 'x is accessible to y'), because it considers a lot more details.

Checkable Artefact Types

A CoCon monitoring tool for a specific artefact type can automatically check the compliance of artefact elements with CoCons if the artefact-type-specific semantics are given in a (semi-)formal language that can be interpreted by the monitoring tool. Artefact-type-specific semantics can only be defined if t he artefact type manages concepts addressed in the artefact-type-independent semantics of the CoCon-predicate. For instance, not every artefact type describes how its elements access each other. The artefact-type-specific semantics of `ACCESSIBLE TO` CoCons can only be expressed for artefact types that address the concept of *communication.*

3.3.6 Formalization of Context-Based Constraints

Predicate Logic

In logic, a *subject* is what we make assertions about, and a *predicate* is what we assert about the subject. According to general convention, subjects are symbolized by lower-case letters, and predicates by capital letters. Hence, the proposition 'the CustomerManagement component is implemented as EJB ' can be translated to '$E(c)$' where '$E()$' symbolizes the predicate 'is implemented as EJB' and 'c' symbolizes the subject 'the CustomerManagement component'. Individual *constants* are symbolized by (lower-case) letters from the front of the alphabet, e.g. a, b, or c. Individual *variables* are symbolized by (lower-case) letters from the end of the alphabet, e.g. x, y, or z. Constants are short names or abbreviations for longer names – 'c' is a constant used to abbreviate 'the CustomerManagement component'. Variables are placeholders that range over individual objects — x in '$E(x)$' is a variable because it specifies no individual on its own, but holds the place for an individual in the universe of discourse. The *universe of discourse*, also called universe, is the set of objects of interest. The propositions in the predicate logic are expressions on objects of a universe. The universe is the domain of the (individual) variables. For instance, it can be the set of all components of a component-based system. There is an unary negation operator ('$\neg$')

and standard connectives are used: conjunction ('$\wedge$'), disjunction ('$\vee$') and implication('$\rightarrow$').

Quantifiers tell us of how many objects the predicate asserts. If we want to assert a predicate of all objects, we use the *universal quantifier* $\forall$ ('for all'). For example, '$\forall x : E(x)$' states that, for all x, x is implemented as EJB; or more idiomatically, all things in the universe of discourse (all the components of a component-based system in this case) are implemented as EJBs. In first-order logic, all variables range over individual objects: all predicate letters are constants, and all quantifiers use individual variables. In higher order logics, variables can be predicates and allow quantification over predicates. *Monadic* predicate logic uses predicates that take just one argument (called one-place predicates). *Polyadic* predicate logic uses predicates (called many-place predicates) that take two or more arguments. One-place predicates assert that their objects have some property or attribute. Many-place predicates assert that their objects stand in some kind of relation. Hence, monadic predicate logic is sometimes called *logic of attributes*, and polyadic predicate logic is sometimes called the *logic of relations*.

Mapping CoCons to Predicate Logic

A CoCon selects two sets: each element of its target set must relate to each element of its scope set as defined by its CoCon-predicate. These CoCon semantics can be expressed via the following predicate logic formula:

$$\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$$

In this formula, the CoCon-predicate is defined via a (polyadic) relation $C(x, y)$, like `x MUST BE ACCESSIBLE TO y`. On the contrary, $T(x)$ and $S(y)$ are monadic predicates on a different level. They define the context condition. As explained in section 3.3.3, they are specified via a query language. $T(x)$ represents the target set context condition, and $S(y)$ represents the scope set context condition. The variable x holds all elements in the target set, and the variable y hold all elements in the scope set. In order to represent a CoCon, $T(x)$ must define a condition on the context property values of x, and $S(y)$ must define a condition on the context property values of y.

The BNF rules of the common CoCon syntax presented in section 3.3.2 are mapped to predicate logic:

- `CoConPredicate` represents the CoCon-predicate $C(x, y)$.
- `TargetSet` represents the context-condition-predicate $T(x)$.
- `ScopeSet` represents the context-condition-predicate $S(y)$.
- The values of `PredicateOperation` represent the following operation on the predicate $C(x, y)$:
  - The terminal symbol `NOT` negates the relation $C(x, y)$ as follows: $\forall x, y : T(x) \wedge S(y) \rightarrow \neg C(x, y)$
  - The terminal symbol `ONLY` is mapped to two propositions:
    * $\forall x, y : T(x) \wedge \neg S(y) \rightarrow \neg C(x, y)$
    * $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$
- The universe of discourse is defined by the values of the `ElementType` rule.

- The values of `ElementName` represent individual constants of the universe of discourse.

Example

For example, an `ACCESSIBLE TO` CoCon defines the CoCon-predicate that each element x in the target set must be *accessible to* each element y in the scope set. By adding the CoCon-predicate operation `NOT`, the semantics are changed: each element x in the target set must be *inaccessible to* each element y in the scope set. According to common BNF rules given in section 3.3.2 and section 3.3.3, the privacy policy example of section 2.2 can be expressed via the following `ACCESSIBLE TO` CoCon:

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

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

- $C(x, y)$ is defined as ‘x is accessible to y’.
- $T(x)$ is defined as ‘x handles Personal Data’
- $S(y)$ is defined as ‘y is used in the Create Report workflow’.
- Finally, the privacy policy is expressed as $\forall x, y : T(x) \wedge S(y) \rightarrow \neg C(x, y)$.

Expressiveness

Is it possible to write down all requirements via CoCons? The simple answer is: No. 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. 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’. 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.

The condition expressed via a CoCon-predicate does not focus on a single type property. For example, a CoCon does not express 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, CoCons CoCon-predicates are limited on expressing how elements having certain type properties *relate to* each other .

Why No Formal Syntax?

CoCons are predicate logic; therefore, their syntax is expressed in predicate logic formulas. Anyone who prefers the $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$ syntax of classic predicate logic can use this syntax in order to define CoCons. However, a requirements specification should serve as a document understood by designers, programmers, and *customers*. Therefore, an easy comprehensible syntax that assists English-speaking persons in understanding the design rationale has been proposed in section 3.3.2.

Different approaches exist for expressing predicate logic in plain English. Any of them can be used instead of the syntax proposed in section 3.3.2.

3.3.7 Formalization of CPQL

In section 3.2.4, context properties have formally been defined. In section 3.3.3, the context property query language CPQL for expressing context condition has semi-formally been defined. This section maps the BNF rules of the CPQL syntax to predicate logic formulas that refer to the context property definition provided in section 3.2.4. According to section 3.3.6, a CoCon can be expressed via the predicate logic formula $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$. Let CV^{cp} be any subset of the valid values of the context property *cp*: $CV^{cp} \subseteq VV^{cp}$. $T(x)$ and $S(y)$ are CPQL expressions. They refer to CV^{cp} sets as follows:

- The BNF rule `ContextCondition` can represent either $T(x)$ or $S(y)$.
- The BNF rule `ContextPropertyName` defines all context property names $\forall l : cp_l \in CP$.
- The BNF rule `ContextPropertyValue` defines all valid values $\forall k, l : v_k \in VV^{cp_l}$
- The terminal symbol '`AND`' represents the conjunction ('$\wedge$').
- The terminal symbol '`OR`' represents the disjunction ('$\vee$').
- The condition '`IS EMPTY`' on those values $values_{cp}(e)$ of the context property *cp* that are associated with the element e is true if $values_{cp}(e) = \emptyset$.
- The condition '$values_{cp}(e)$ `CONTAINS` CV^{cp}' is true if $values_{cp}(e) \subseteq CV^{cp}$.
- The condition '$values_{cp}(e)$ `DOES NOT CONTAIN` CV^{cp}' is true if $values_{cp}(e) \nsubseteq CV^{cp}$.
- The condition '$values_{cp}(e)$ `EQUALS` CV^{cp}' is true if $values_{cp}(e) = CV^{cp}$.
- The condition '$values_{cp}(e)$ `DOES NOT EQUAL` CV^{cp}' is true if $values_{cp}(e) \neq CV^{cp}$.
- The condition '$values_{cp}(e)$ `INTERSECTS WITH` CV^{cp}' is true if $values_{cp}(e) \cap CV^{cp} \neq \emptyset$.
- The condition '$values_{cp}(e)$ `DOES NOT INTERSECT WITH` CV^{cp}' is true if $values_{cp}(e) \cap CV^{cp} \neq \emptyset$.

Formalization Summary

This section merges the definition of CoCons (in this section) and the definition the context properties (in section 3.2): it defines how to refer to *context property values* in a *context condition* of a *CoCon*. Hence, a formal basis has been provided for each of the three replaceable conceptual modules depicted in figure 3.4 on page 33. These formal bases have been defined in an artefact-type independent way.

3.3.8 Comparing Context-Based Constraints with Aspects

According to [KLM+97], **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. Even though CoCons can be implemented via aspect-oriented frameworks, CoCons add the new notion of context-based point cuts as explained next and in [Büb05].

Join Cuts

A CoCon-predicate $C(x, y)$ expresses a crosscutting concern. In terms of AspectJ ([Asp]), $C(x, y)$ expresses an **advice**. A place where to weave in an advice is called **join point** in AspectJ. A **pointcut** defines the conditions under which to weave in an advice – it defines a query for selecting the join points. According to [SHU04], defining the pointcuts is independent design issue and can be accomplished separate from other tasks such as modelling the advices. One advice affects several parts of a software system. Handling crosscutting concerns via aspect-oriented programming is well understood at source-code level or at runtime. However, it is difficult to recognize or 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.

Typically pointcuts select their 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, my goal is to define pointcuts without knowing the source code. I suggest to express aspects in a way that is understandable for stakeholders and customers by using a new notion of weaving. CoCons determine where to weave in which aspect by considering the system's context – their pointcuts are context-based.

CoCons express metadata-based aspects. But, CoCons stick to special notions of pointcuts as discussed next and in [Büb05]. 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 criteria in this condition refer to details that are part of the source code – a pointcut query quantifies over properties of the source code.

All AOP systems provide a language to define pointcuts. 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.

Context-Based Point Cuts

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 3.2.1, the context doesn't have to be part of the source code or managed by the system. Likewise, [Lad05] 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.

Aspect-oriented software engineers have to define the relationships between aspects and their target artefacts. I suggest using context as glue between advices and joining points for two reasons. Contexts are implementation-independent and maybe 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 pattern 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 2.2 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.

CoCons vs. 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. In [Lad05], several mechanisms are examined for referring to metadata in pointcuts. In [SP02], 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 [LBBK03, Rat04] because we wanted our frameworks to work without modifying the components.

CoCons vs. Aspects at Design Level

With regards to considering aspects already during design, several interesting approaches exist. In [AMBR02], cross-cutting 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 [SHU04]. 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 [TOHJ99], a hyperslice encapsulates a cross-cutting 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 reflecte in the 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 [Cla02]: 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.

CoCons vs. Aspects at Requirements Level

According to [BK05], 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 Theme approach described in [BC04] 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 [KR04] 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.

3.3.9 Research Related to Context-Based Constraints

Adaptive Approach

The 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 system 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 are *adaptive*: they can adapt automatically to changed system elements or to changed contexts because their constrained elements are identified newly each time the compliance of the system with the CoCon is checked.

Goals

Goal-oriented requirements engineering is well established. According to [vL01]), goals denote the objectives a system must meet. Eliciting goals focuses on the problem domain and the needs of the stakeholders, rather than on possible solutions to those problems. An important benefit of expressing high-level goals via CoCons is that a CoCon can indirectly select its constraint elements according to their context and, thus, trace a high level goal to the system elements where the goal is operationalised. 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 with the system parts involved. It can be unknown which element are involved in the goal when writing it down as a CoCon.

Traceability Links

According to [Pin00], 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 [RJ01], 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.

Adaptive Traceability

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 [RJ01], 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.

'Ilities'

Recent work by both researchers ([CNYM00]) and practitioners ([RR99]) has investigated how to model non-functional requirements and to express them in a form 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. Most of the CoCon-predicate proposed in this thesis specify non-functional requirements. Via the two-step approach for defining the semantics, these CoCon-predicates can express 'ilities' clearly. They are particularly helpful in expressing crosscutting ilities that apply to more than one system element, because one CoCon can constrain several involved elements according to their context property values.

xlinkit/CLIX

The first order logic based rule language CLIX is now compared with CoCons. The xlinkit framework as presented in [NEF01] 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 of the same artefact, while context conditions don't necessarily refer to the artefact elements of the same artefact. 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 3.3.6, 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: one CLIX rule applies to artefacts of one type, while one 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.

Quick Tour $\xrightarrow{goto}$ Chapter 4

Section 3.4 will discuss defining *when* to check a CoCon and *what to do* if the CoCon is violated. Quick readers can skip this discussion of action semantics and proceed with chapter 4 on page 53. It will discuss how to automatically detect requirement violations if the requirements have been written down via CoCons.

3.3.10 The Fundamental Things Apply As Time Goes By

CoCons have been invented in 2000. In the meantime, some papers discussing CoCons have been published. Unfortunately, the concepts stayed the same, but their names changed. This section explains, which old names should not be used anymore. If you did not read the old papers, skip this section.

‘Instructions’ became ‘Constraints’

The most obvious change happened early: in [Büb00a] and [Büb00b], context-based ‘constraints’ were called context-based ‘instructions’. All later publications use the term context-based ‘constraints’ (CoCons). Sometimes, I used the term **CoCon type** instead of ‘CoCon-predicate’ in order to explain the concepts to people without mathematical background: some ‘types’ of CoCons express security requirements, while other CoCon-predicates address other requirement issues. Different CoCon-predicates have been proposed. Sometimes, the same CoCon-predicate has been renamed. `NOT ACCESSIBLE TO` CoCons, for instance, have been called `InAccessibleBy` CoCons. Their semantic, however, stayed the same.

Moreover, the event and action semantics addressed in early CoCon papers have been separated from the CoCons and are now expressed via CoCon-Rules as described in the next chapter.

3.4 Turning CoCons into CoCon-Rules by Adding Events And Actions

3.4.1 Introduction to Business Rules and Policies

(Business) Rule

Systems analysts have long been able to describe an enterprise in terms of the structure of the data that enterprise uses and the organization of the functions it performs, but they have tended to neglect the constraints under which the enterprise operates. Frequently these are not articulated until it is time to convert the constraints into program code. While rules that are represented by the structure and functions of an enterprise have been documented to a degree, others have not been articulated as well, if at all. A **(business) rule** is an expression that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behaviour of the business. It prevents, causes, or suggests things to happen.

Policy

A collection of rules is called policy. Requirements can be recorded via policies. Policies are rules governing the choices in behaviour of a system according to [Slo94].

Large-scale systems may manage many users and resources. In that case, it is not practical to specify policies relating to individual elements. Instead, it must be possible to specify policies relating to groups of elements. CoCons can specify rules for possibly large groups of elements.

3.4.2 Difference between CoCons and CoCon-Rules

CoCon-Rule = Event(s) + CoCon + Action(s)

The previous sections discuss constraints. A constraint is an assertion ([Mey88]), not an executable mechanism. It does do not tell what happens when it is violated or when the system complies with the constraint. An expression that defines what to do if a condition is met or violated is called a rule. **Event Condition Action** (ECA) rules are a well-known approach for specifying rules. According to [WD93], they are typically applied in active databases. An ECA rule consists of three parts: the *event* that triggers the validation of the system for compliance with the rule’s *condition*. This condition can be defined by a constraint. A **CoCon-Rule** is an ECA rule whose condition is defined by a CoCon. Moreover, an *action* can be specified that must be taken if the rule’s condition is (not) met. However, the *action* of a rule can result in side effects as discussed next. A CoCon-Rule consists of a CoCon and additionally can refer to events or actions. Therefore, a CoCon-Rule has almost the same syntax as its CoCon. As long as the expression does *not* refer to events or actions it is called context-based constraint or CoCon. As soon as it refers to events or actions it is called CoCon-Rule. A CoCon-Rule can define when to check the system for compliance with the CoCon and what to do if the CoCon is violated or if the system complies with the CoCon. After explaining the difference between CoCons and CoCon-Rules in section 3.4.2, the following sections define the semantics and syntax of CoCon-Rules.

3.4.3 Limitations of Enriching CoCons with Actions and Events

Butterfly Effect

A CoCon can be validated due to an event, and the result of its validation can call for certain actions. However, not every reason for validating a CoCon can be considered in a CoCon-Rule. Moreover, not every action resulting from the CoCon validation should be stated in a CoCon-Rule. The term chaos, with reference to chaos theory, refers to an apparent lack of order in a system that nevertheless obeys particular laws or rules. The two main components of chaos theory are the ideas that systems - no matter how complex the may be - rely upon an underlying order, and that very simple or small systems and events can cause very complex behaviours or events. This latter idea is known as *sensitive dependence on initial conditions*, a circumstance discovered by Edward Lorenz in the early 1960s. The butterfly effect, first described by Lorenz at the December 1972 meeting of the American Association for the Advancement of Science in Washington, D.C., vividly illustrates the essential idea of chaos theory in his talk “Predictability: Does the Flap of a Butterfly’s Wings in Brazil set off a Tornado in Texas?”. The example of such a small system as a butterfly being responsible for creating such a large and distant system as a tornado in Texas illustrates the impossibility of making predictions for complex systems; despite the fact that these are determined by underlying conditions, precisely what those conditions are can never be sufficiently articulated to allow long-range predictions.

Directly Related Events

A butterfly in brazil can cause a tornado in Texas or the violation of a CoCon. But, not *every* cause for the CoCon violation can be considered in a CoCon-Rule. For instance, a CoCon-Rule should not state to check its CoCon ‘when a butterfly flaps in brazil’, even if the butterfly indirectly causes the modification of the component. Instead, a CoCon-Rule should only refer to those events that directly trigger the validation of its CoCon. For instance, a CoCon-Rule can state to check its CoCon ‘when a component is modified’.

Side-Effects

Actions can result in unwanted side effects. For example, an action can cause a tornado in Texas. Or it can remove one of the system’s components. Performing actions can result in unreliable systems. On the contrary, expressions in a declarative constraint language cannot have side effects. The state of a system does not change because of evaluation of a constraint. The person who specifies the constraint does not decide how the violation of the constraint should be handled. This facilitates to divide and conquer problems. It results in a clear separation between specification and implementation. Therefore, this thesis focuses on constraints.

Atomicity

A CoCon-Rule can demand to take an action according to the rule’s condition. when a CoCon-Rule is checked with the aim to perform an action on it, the check and its corresponding action must be regarded as one atomic action. If the context property values on which the CoCon-Rule depends change during the evaluation of the CoCon's action, the outcome of that check is not reliable.

3.4.4 Referring to Events and Actions in CoCon-Rules

Events

An event is a noteworthy occurrence according to [OMG03b]. A CoCon-Rule can state on which event the compliance of the system with the CoCon is checked. For example, an event that possibly changes the context property values that are checked in the CoCon's context condition can be named in the corresponding CoCon-Rule. Only events that can be detected by the software system should be considered, because otherwise the CoCon-Rule cannot be monitored automatically. Hence, a CoCon-rule should refer to names of events that are raised by applications.

The two new CoCon attributes described here define actions that must happen if a CoCon is (or is not) violated. Each attribute has a name and one or more value(s).

`COMPLIANCE-ACTION`

A CoCon can be turned into a CoCon-Rule by adding the attribute **COMPLIANCE-ACTION**. It describes what action must be taken if two elements are related via this CoCon and comply with the CoCon-predicate. Its value depends on the software system artefact that is checked for compliance with the CoCon.

`VIOLATION-ACTION`

The attribute **VIOLATION-ACTION** describes what action must be taken if two elements are related via this CoCon but don't comply with the CoCon-predicate. Its value depends on the software system artefact that is checked for compliance with the CoCon. Two general values of `VIOLATION-ACTION`s are proposed next that do not depend on the artefact type or the abstraction level:

- The value '**Ignore Conflicting Elements**' of the `VIOLATION-ACTION` attribute defines that the system ignores the elements that violate a CoCon.
- The value '**Warn**' of the `VIOLATION-ACTION` attribute defines that the user is notified of detected violations of a the CoCon. More precise values of `VIOLATION-ACTION`, e.g. 'warn system designers via a pop up window and a fire-alarm sound' can be used to define who shall be notified and how the person(s) shall be notified.

`IF-THEN-ELSE`

In regards to `IF-THEN-ELSE` expressions, the `COMPLIANCE-ACTION` is defined in the `THEN` part, and the `VIOLATION-ACTION` is defined in the `ELSE` part. Before defining the syntax for turning a CoCon into a CoCon-Rules, general limitations of the defining events and actions are discussed next.

3.4.5 The Common CoCon-Rule Syntax

As explained in section 3.4.2, the syntax of CoCon-Rules is almost the same as the CoCon syntax defined in section 3.3.2. Only two BNF rules are added to the syntax definition:

Common Syntax of CoCon-Rules

`CoConRule`	::=	[‘**ON**’`(ValidationEvent)`$+^{AND\ \mid\ OR}$] `CoCon` [‘**-**’ `(ActionAttribute)`$+^{AND}$]
`ActionAttribute`	::=	`ActionAttributeName` ‘=’ `(AttributeValue)`$+^{Comma}$
`ActionAttributeName`	::=	‘**COMPLIANCE-ACTION** ’| ‘**VIOLATION-ACTION** ’

Example A revisited

According to common BNF rules given here, the privacy policy of section 2.2 can be turned into a CoCon-Rule by adding an action as follows:

```
ON Dial-Up ALL COMPONENTS WHERE ‘Personal Data’ = ‘True’ MUST
NOT BE ACCESSIBLE TO THE COMPONENT ‘WebServer’- VIOLATION-ACTION
= ‘Block Call’ .
```

4. Applying CoCons in Continuous Software Engineering

Structure of this Chapter

This chapter discusses using CoCons after they have been written down. First, section 4.1 explains why to consider requirements as invariants. The next sections discuss how to detect violated or contradicting CoCons *automatically*. Two different kinds of detectable conflicts are discussed: an CoCon-violation conflict occurs if an artefact element does not comply with a CoCon's CoCon-predicate on how it must (or must not) relate to another artefact element as discussed in section 4.2. Moreover, an inter-CoCon conflict occurs if one CoCon contradicts another CoCon as discussed in section 4.3. Next, section 4.5 introduces concepts for maintaining context property values in system modifications because violated or contradicting CoCons cannot be detected at all if they refer to wrong, outdated, or missing context property values. Finally, section 4.4 outlines proof-of-concept software tools.

4.1 Continuous Requirements Tracing

A software system is described in analysis level artefacts, in design level artefacts, in source code level artefact, and in runtime artefacts. When modifying one artefact, the system's artefacts should stay consistent on all abstraction levels. New methods and techniques are required to safely transform the system's artefacts without the unintentional violation of existing dependencies or invariants. Absolute consistency, however, can hardly be achieved because time pressure and limited resources often prevent keeping track of *all* dependencies and invariants across *all* artefacts on *all* abstraction levels when modifying the system. In spite it isn't possible to anticipate *all* consequences of a change, the consideration of dependencies and invariants *improves* consistency in modifications. If one element of one artefact at one abstraction level is modified, then this initial modification can have an impact on other elements both in the same artefact and in other artefacts. Continuous software engineering aims to lessen the chaos that results from 'blind' modifications as described by Parnas in [Par94]:

Changes Can Cause Chaos

> "Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one must know both the original design rules, and the newly introduced exceptions to the rules, to understand the product. After many such changes, the original designers no longer understand the product. Those who made the changes, never did. In other words, nobody understands the modified product."

Change Propagation

Successful software systems always evolve as the environment in which these systems operate changes and stakeholder requirements change. Therefore, *managing change* is a fundamental activity in requirements engineering. Change impact analysis is defined in [AB93, AB98] as 'the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change'. The prevailing work on change impact analysis focuses on the *source code level* of *not*-component-based systems. On the contrary, continuous software engineering focuses on *models* of component-based systems.

According to [MB99], dependencies can be numerous for complex systems. Many kinds of existing dependencies can be obtained automatically from source code via compiler theory, such as dependency analysis or program slicing. For instance, figure 4.1 shows a screenshot published in [MB99] that depicts only an excerpt of only the *call* dependencies in a small software system consisting of 15 classes and 2000 lines of code.

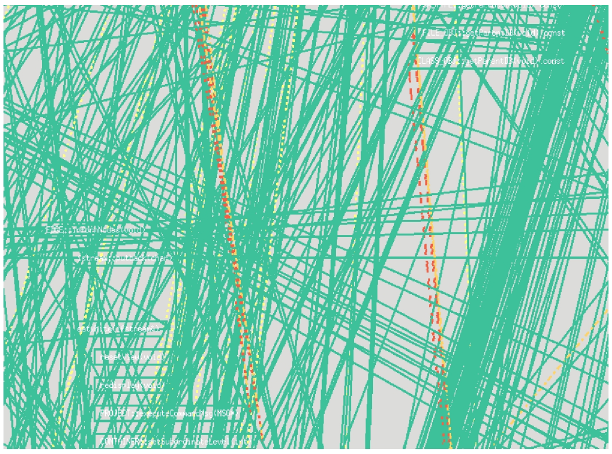

Figure 4.1: Excerpt of a Call Graph Depicting Dependencies Between 15 Classes

Domino Effect

A change can result in many consequences because of to the 'ripple' ([AB98, QV94]) or 'domino effect' ([Ber02]): due to dependencies, an initial change at one place can require changes at other places which in turn require changes in other places and so on. As soon as the dependent element is modified, this modification might induce other changes at other places. A sequence of induced modifications is called **propagation path**. If many dependencies exist, the propagation path of an initial modification can get long. The change impact of an initial modification increases by the length of the resulting propagation path because more components must be modified if the propagation path is longer. Modifying many artefacts is expensive. Moreover, a common problem in reengineering is the introduction of new faults as a side effect of a modification. Hence, the domino effect of change propagation should be stopped as soon as possible.

Continuous Requirements Tracing

Section 3.3.9 has discussed that CoCons are traceability links. Hence, they facilitate to monitor and control the impact of changes. According to [Pal00], "traces allow to identify, when objectives change, which parts of the system are still relevant, which are not". CoCons can indirectly select their constrained elements. This facilitates identifying 'which parts of the system are still relevant' because the person that writes down the changed objective via a CoCon does not need to know each relevant element when writing down the CoCon. Instead, only the context in which the change takes place must be known in order to identify the constrained elements.

CoCons define invariants for possibly large sets of elements. Change impact can generally be reduced by defining invariants at many places because each single invariant can stop change propagation. One CoCon can constrain many artefact elements. Hence, one CoCon can stop change propagation at many places.

Changes to requirements specifications include adding, changing or deleting requirements. If a requirement is specified via a CoCon and this requirement changes then the CoCon has to be changed. It is much easier to change one CoCon expression then to change all the artefact-type-specific invariants at all the corresponding elements of all artefacts involved in the development process. Only one expression (the artefact-type-independent CoCon) must be adapted if the corresponding requirement changes in order to be able to check the model, the configuration files, or the components at runtime. Moreover, the requirement expression doesn't have to be re-written if a different artefact type (or version) is used. According to [GM78], the ability to allow changes to any artefacts to be traced throughout the system is an important property of any system description technique. An invariant specified via a CoCon can be checked for different modelling languages, different programming languages, different component models or different component platforms.

Requirements traceability is intended to ensure *continuous* alignment 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 or not. As discussed above, limited resources typically prevent achieving absolute consistency. A tool that facilitates to detect which artefact elements don't comply with which requirements helps to improve consistency, though. Tools that can be applied without much effort support **continuous requirements tracing** better because the system artefacts can be checked more frequently if each check takes less effort. It takes much effort to associate each element with the relevant context properties. Therefore, section 4.5 will discuss how to maintain context property values in order to reduce this effort. If the context property values are available then CoCons facilitate continuous requirements tracing because they can automatically identify the elements involved in a change. They reduce the effort for locating those elements that don't comply with the requirements.

In order to support continuous requirements tracing via tools, two different ways of checking a system for compliance with CoCons are examined in this chapter. First, section 4.2 discusses algorithms for identifying those artefact elements that violate a CoCon. Then, section 4.3 suggests an algorithm for detecting whether one CoCon contradicts other CoCons.

4.2 Detect CoCon Violations

This section examines how to detect violated CoCons automatically.

4.2.1 CoCon-Violation Conflicts

As explained in section 3.3.6, a CoCon can be expressed via the predicate logic expression $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$. In this expression, the CoCon-predicate $C(x, y)$ defines how the constrained artefact elements x and y must relate to each other. A **CoCon-violation conflict** occurs if the relation between the element x to another artefact element y does not comply with the CoCon-predicate $C(x, y)$. For instance, the privacy policy example given section 2.2 demands that certain x must be inaccessible to certain y. The next section presents an algorithm that enables software tools to monitor the system artefacts for CoCon violations.

4.2.2 The Detect-CoCon-Violations Algorithm

This section presents a general algorithm for identifying CoCon-violation conflicts. After presenting the algorithm, its complexity is discussed.

```
input : The finite set $A$ containing all $n$ elements $a_1, ..., a_n$
        of the checked artefact and the CoCon
        $\forall x, y \in A : T(x) \wedge S(y) \rightarrow C(x, y)$
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 scope set context condition $S(a_i)$ then
        add $a_i$ to the scope set $SCOPE$
    if $a_i$ fulfils the target set context condition $T(a_i)$ then
        add $a_i$ to the targe set $TARGET$
/* 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 that shall be checked and the CoCon that shall be checked for whether 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 whether it violates $C(x, y)$. 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 $C(x, y)$. If the algorithm returns an empty result set of pairs R then no conflicts were found. Otherwise, R contains the pairs of those elements which violate the CoCon.

Complexity

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 $S(x)$ and $T(x)$ 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 evaluates 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 according to $Semantics^{C(x,y)}_{Artefact-Type}$ has a linear complexity.

Pitfalls However, the complexity of algorithm 1 can grow out of hand for two reasons. Checking the context conditions $S(x)$ or $T(x)$ can have high complexity if they refer to a complex context model and if the query capabilities or their query language are unrestricted. Hence, I suggest using context models and context conditions that together result in linear complexity. In regards to context models, I decided not to consider the context of context (...of context) as explained in section 3.2.1 because queries to such a recursive context model might have undecidable complexity. Furthermore, section 4.5 will discuss which query capabilities are not part of CPQL in order to stick to context conditions checks of linear complexity.

Additionally, checking the constrained elements for whether they fulfil the CoCon-predicate can have high complexity if its artefact-type-specific $Semantics^{C(x,y)}_{Artefact-Type}$ are complex. In order to avoid complex CoCon-predicates, I propose only using *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 as explained next. A CoCon-violation conflict can only be detected automatically if the *artefact-type-specific* semantics of the checked CoCon-predicate are defined and computable. For instance, in order to check a UML model for whether it complies with the privacy policy CoCon, the artefact-type-specific $Semantics^{x-is-accessible-to-y}_{UML-2.0}$ must be defined as done in chapter 6. Only those $Semantics^{C(x,y)}_{Artefact-Type}$ conditions should be used whose computation has linear complexity because $Semantics^{C(x,y)}_{Artefact-Type}$ will be computed $s \times t$ times in the 'check constrained elements' loop of the Detect-CoCon-Violations algorithm. The discussion on preventing non-linear complexity in artefact-type-specific $Semantics^{C(x,y)}_{Artefact-Type}$ definitions will continue in section 6.2.1 where an example is discussed in detail.

In regards to continuous software engineering, CoCons facilitate considering invariants in system modifications because software tools can use algorithm 1 to automatically detect each artefact element that doesn't comply with the CoCons.

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

4.3 Detect Contradicting CoCons

4.3.1 Inter-CoCon Conflicts

The CoCons or context properties of a system can change over time. A new CoCon can be added to the system or an already existing CoCon can be modified. This new or modified CoCon can contradict prevailing CoCons. Moreover, each changed context property value can result in contradicting CoCons, because a CoCon can newly (not) constrain the element having the changed context property value. A conflict between constraints arises if they express opposite conditions on the same elements. Likewise, an **inter-CoCon conflict** occurs if two CoCons contradict each other. This section discusses how to detect conflicts between CoCons in order 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* $C(x, y)$. As explained in section 3.3.6, a CoCon can be expressed via the predicate logic formula $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$.

$NOP \leftrightarrow NOT$ Inter-CoCon Conflict

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

- $\forall x, y : T_1(x) \wedge S_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : T_2(x) \wedge S_2(y) \rightarrow \neg C(x, y)$

contradict each other if $\exists x, y : T_1(x) \wedge T_2(x) \wedge S_1(y) \wedge S_2(y)$.

Example

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 2.2 can be expressed in CCL 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’.

$NOP \leftrightarrow ONLY$ Inter-CoCon Conflict

The next two inter-CoCon conflict types take the CoCon-predicate operation `ONLY` into account. According to section 3.3.6, the semantics of the operation `ONLY` are expressed in *two* formulas: $\forall x, y : T(x) \wedge \neg S(y) \rightarrow \neg C(x, y)$ and $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$. 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

- $\forall x, y : T_1(x) \wedge S_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : T_2(x) \wedge \neg S_2(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_2(x) \wedge S_2(y) \rightarrow C(x, y)$

contradict each other if $\exists x, y : T_1(x) \wedge T_2(x) \wedge S_1(y) \wedge \neg S_2(y)$.

Example

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.

$ONLY \leftrightarrow ONLY$ Inter-CoCon Conflict

$ONLY \leftrightarrow ONLY$ inter-CoCon conflicts can occur if one CoCon with the CoCon-predicate operation `ONLY` contradicts another CoCon with the CoCon-predicate operation `ONLY`:

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

- $\forall x, y : T_1(x) \land \neg S_1(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_1(x) \land S_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : T_2(x) \land \neg S_2(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_2(x) \land S_2(y) \rightarrow C(x, y)$

contradict each other if $\exists x, y : T_1(x) \land T_2(x) \land ((\neg S_1(y) \land S_2(y)) \lor (S_1(y) \land \neg S_2(y)))$.

Example

If $C(x, y)$ is defined as `x MUST BE ACCESSIBLE TO y` CoCon-predicate then the $ONLY \leftrightarrow ONLY$ inter-CoCon conflict states that no element x must be `ONLY ACCESSIBLE TO` y_1 if x is `ONLY ACCESSIBLE TO` any y_2 and y_2 is not part of the same scope set as y_1. For instance, following CoCon can cause a $ONLY \leftrightarrow ONLY$ inter-CoCon conflict:

CoCon 4: : `ALL COMPONENTS WHERE ‘Personal Data’ CONTAINS ‘True’ MUST ONLY BE ACCESSIBLE TO ALL COMPONENTS WHERE ‘Operational Area’ CONTAINS ‘Sales’`

CoCon 4 contradicts CoCon 3 if a component y that is used *either* by the controlling department *or* by the sales department is accessible to a component x that handles personal data. – as long as y is not used in both departments, one of the two CoCons is violated.

‘NOT-SELF’ 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 $\forall x, y : T(x) \land S(y) \rightarrow \neg C(x, y)$ contradicts the proposition $\exists x : T(x) \land S(x)$.

Example

`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 these four *general* inter-CoCon conflicts, *CoCon-type specific* inter-CoCon conflicts exist as listed in chapter 5.

4.3.2 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.

```
input : The finite set $A$ containing all $n$ elements $a_1, ..., a_n$
        of the checked artefact and the two CoCons
        $\forall x, y \in A : T_1(x) \wedge S_1(y) \rightarrow C(x,y)$ ($CoCon_1$) and
        $\forall x, y \in A : T_2(x) \wedge S_2(y) \rightarrow \neg C(x,y)$ ($CoCon_2$)
output: The set $SCOPE$ containing those scope set elements that cause
        a conflict between $CoCon_1$ and $CoCon_2$ and set $TARGET$ con-
        taining those target set elements that cause a conflict between
        $CoCon_1$ and $CoCon_2$
/* check $\exists x, y : T_1(x) \wedge T_2(x) \wedge S_1(y) \wedge S_2(y)$ */
foreach $a_i \in A$ do
    if $S_1(a_i)$ and $S_2(a_i)$ then
        add $a_i$ to the set $SCOPE$
    if $T_1(a_i)$ and $T_2(a_i)$ 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

As input, algorithm 2 needs the artefact that shall be checked and the two CoCons that shall be checked for whether 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 4.3.1 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 $CoCon_2$. This result allows us to trace which element is inconsistent to which other element. As in the xlinkit framework (see section 3.3.9), this result could be expressed as hyperlinks between inconsistent elements.

Complexity The set of artefact elements A contains n elements. Hence, $4n$ context condition checks happen in the 'check $\exists x, y : T_1(x) \wedge T_2(x) \wedge S_1(y) \wedge S_2(y)$' loop of algorithm 2. If each context condition check has linear complexity than the overall complexity of algorithm 2 is $O(n)$.

More Algorithms Algorithm 2 only covers the first inter-CoCon conflict listed in section 4.3.1. 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 4.3.1.

Enhancements

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 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 condition 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 4.2.2.

Algorithm 1 vs. 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. Inter-CoCon conflicts between two CoCons, however, can be detected without artefact elements if the scope context conditions of both CoCons are identical and the target context conditions of both CoCons are identical. In that rare case, these equal context conditions will always select the same constrained elements. Therefore, an inter-CoCon-conflict can be predicted. In most cases, two CoCons of one system will not have equal context conditions for both their scope sets and their target set, though. Hence, most inter-CoCon conflicts can only be detected if the CoCons constrain any elements at all.

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: Checking the artefact-specific $Semantics^{C(x,y)}_{Artefact-Type}$ for each pair of constrained elements reduces the overall performance of algorithm 1. Furthermore, algorithm 1 simply cannot be run if $Semantics^{C(x,y)}_{Artefact-Type}$ is unknown. 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.

4.4 Proof-of-Concept Tools

According to [Pal00], 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, monitor them for compliance with the CoCons and notify the designer if an element violates a CoCon or if a CoCon contradicts another CoCon:

Three prototypical proof-of-concept tools for monitoring software system artefacts for compliance with CoCons exist:

- The open source CASE tool 'ArgoUML' already has a model-validation mechanism called design critics ([RR98]). The CCL-plugin ([Ski01, vA03, Len03]) adds design critiques to ArgoUML that check the compliance of a UML model with CCL expressions. The artefact-type-specific semantics of CCL for UML models is defined in chapter 6 via OCL.
- The 'EJB-Complex' framework presented in [LBBK03, Bil02, BL02, LB02, Wan02] can validating 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. Moreover, it uses 'dynamic proxies' for intercepting method invocations. 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 [Rat04]. JBoss AOP and AspectWerkz support metadata in their current versions. The upcoming version of AspectJ will support metadata by modifying the AspectJ language.

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.

4.5 Maintaining Context Property Values

A CoCon can refer to the context property values of an element in order to check whether the element is constrained by the CoCon. This indirect selection of constrained elements fails if the context property values of an element are wrong or missing. Hence, it must be ensured that the context property values are always up-to-date and truthful. One approach for preventing illegal context property values has been introduced in section 3.2.4: the set of valid values (VV^{cp}) must be defined for a context property *cp*. However, there are many reasons why even a valid context property value can be wrong: it can be outdated, wrong due to incompetence, or even wrong with ill intent. The next sections discuss mechanisms for improving the trustworthiness of context property values.

4.5.1 Type-Instance Constraint On Context Property Values

Types & Instances

In UML ([OMG03b]), a **type** is a (stereotyped) class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type is a blueprint of possibly many physical entities - these entities are called **instance** of the type if their properties comply with their type definition. In UML, an 'instance' is an entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations.

Type Values Determine Instance Values

The value of a context property can both be associated with a type or its instance. For instance — oops, sorry: for example, the value can be associated with a component type or with instances of this component type. The context property values associated with an element type, however, are not automatically associated with the instances of this element type (or vice versa) because not all instances will be in situated in the same situations. This section suggests how to relate the type context with the context of the type's instances.

Type-Instance Constraint

The **type-instance constraint on context property values** applies if the element $e_{instance} \in E$ is an instance of another element $e_{type} \in E$:

$$values_{cp}(e_{instance}) \subseteq values_{cp}(e_{type})$$

Only the values $v_{1..n}$ associated with e_{type} are allowed to be associated with $e_{instance}$. These values $v_{1..n}$ are a subset of the values in VV^{cp} because only values in VV^{cp} can be associated with e_{type}.

Example

For example, 10 different workflows called A, B, C, D, E, F, G, H, I, and J may be performed by a system. Thus, context property name *cp* = 'Workflow' has the valid values $VV^{Workflow}$ = {A, B, C, D, E, F, G, H, I, J }. If the designer associated only 5 of the 10 valid values with the component type e_{type} = 'WebServer' via $values_{workflow}(e_{type})$ = {A, B, C, D, E} then it is not allowed to associated the instance $e_{instance}$ of this component type with F, G, H, I, or J. At runtime, the context property value of $e_{instance}$ must be a subset of the type's values: $values_{workflow}(e_{instance}) \subseteq$ {A, B, C, D, E}.

New: Type Context

None of the context models listed in section 3.2.1 distinguish between the context of types and the context of their instances. The type-instance constraint on context property values defines a new notion of **type context**: the context property values associated with a type characterize the possible situations in which the type's instances are (or will be) situated.

Managing Change

If a context property value newly is associated with any instance but has not been associated with the corresponding type yet then the type-instance constraint is violated: the new context property value either cannot be associated with the instance, or it has to be associated with the type, too. Hence, the type-instance-constraint assists in keeping the context property values of a model and its corresponding runtime system consistent when modifying either the runtime system or the model.

4.5.2 Dependent Context Property Values

The context property value associated with an element can depend on other influences. For example, a value can depend on the current state of the associated element at runtime, or on other context property values associated with the same element. These **dependent context property values** are explained now, and a notation is defined for them.

Extracting Context

If we know on which information the current context of an element depends than we can determine this context of this element - we can automatically extract the context from those places where the information is stored on which it depends. If the current value of an element's context is extracted automatically every time when the dependent context is queried then nobody needs to set the context value manually. It will always be available and up to date as long as the information on which it depends is available and up to date.

Syntax

If a context property value associated with an element depends on something, this dependency is written down in round brackets behind the context property value's association to the element. If several context property values associated with the same element depend on the same, they are grouped within curly brackets. BNF Rules concerning separators (‘,’(Comma), ‘OR’ or ‘AND’) are abbreviated: “`Rule { Separator Rule }*`” is abbreviated as “`(Rule)`+^{Separator}”.

The Syntax for Dependent Context Property Values

ContextPropertyValue	::=	(OneOrMoreValues [(‘**(IN STATE**’ NameOfState+^{Comma} ‘**)**’) | (‘**(IF**’ BooleanCondition+$^{(AND \mid OR)}$ ‘**)**’)]) | (‘**?(CALCULATED VIA**’ Algorithm ‘**)**’)
OneOrMoreValues	::=	Value | (‘**{**’ (Value)+^{Comma} ‘**}**’)
BooleanCondition	::=	ContextCondition | BooleanConstraint

Semantics

A dependent value associated with an element is **unused** if it its dependency doesn't hold. Examples are discussed in this section. If the value is **in use** then it reflects the current context of the element. If it is unused, it does not reflect the current context of the element because its dependency does not hold. This syntax given above can express different types of context property value dependencies. They are introduced now via examples for an element e:

State-Dependent

State-dependent context property values are defined via the terminal symbol `IN STATE`. Other specification techniques, e.g. state diagrams or Petri nets, are needed to define the values of `NameOfState`. If a context property value is specified without a dependency, e.g. ‘Sell Product’ in the following example, it is always associated with e regardless of e's state:

```
CurrentWorflow(e): ‘Add New Customer’(in state s4,s5), {‘Delete
     Customer’, ‘Modify Customer’}(in state s1), ‘Sell Product’
```

Condition-Dependent

Condition-Dependent context property values are in use if the `Boolean-Condition` evaluates to true. Two different kinds of boolean conditions are proposed here.

On the one hand, `BooleanConstraint` can refer to the current state of an element via a Constraint. In contrast to the syntax for state-dependent values explained above, the state of an element is not identified by the state's name here. Instead, the boolean constraint defines a condition on which the associated element is checked every time the current value of this context property for this element is queried. If the associated element is an UML model element then the boolean constraint can be specified via the Object Constraint Language OCL . For example, the current context property value can depend on the attribute age of the element *e*:

```
CurrentWorflow(e): ‘Add New Customer’(if self.age < 18), {‘Delete
     Customer’, ‘Modify Customer’}(if self.age ≥ 18)
```

On the other hand, the boolean condition can refer to other context property values associated with the same element. Details on context conditions, e.g. the `ContextCondition` syntax rule, have been defined in section 3.3. A **context-condition-dependent** context property value is only *in use* if the context condition matches to the other context property values associated with the same element *e*. In the following example, the value ‘Modify Customer’ is context-condition-dependent:

```
CurrentWorflow(e): {‘Delete Customer’, ‘Add New Customer’(if UserRole
     EQUALS ‘Controller’), ‘Modify Customer’}(if UserRole EQUALS
     ‘Trader’), ‘Sell Product’
```

The value ‘Modify Customer’ only is in use if *e* is also associated with the value ‘Trader’ of the context property ‘UserRole’.

Calculation-Dependent

The current state of an element can also be defined via an algorithm expressed in the `Algorithm` rule. **Calculation-dependent** context property values are calculated by the tool that checks the system for compliance with the CoCon according to the given algorithm. The algorithm is given within round brackets starting with `CALCULATED VIA` and refers to the attributes or methods of the associated element *e* or other elements that are associated with *e*. The algorithm defines how to calculate the value, e.g. in the following example using OCL:

```
CurrentWorflow(e): ?(CALCULATED VIA self.GetWorkflowname()),
     ‘Modify Customer’
```

In this example, the current value of the context property‘CurrentWorkflow’ is extracted by invoking the method ‘GetWorkflowname()’ of *e*. Calculation-dependent values are similar to state-dependent values. The algorithm refers to the *state* of instances of the associated element(s). In contrast to state-dependent values, the value resulting of the algorithm is not pre-defined. In the condition-dependency example given above, the value ‘Add New Customer’ is predefined and is *in use* if the constraint ‘self.age < 18’ is true. On the contrary, the value returned by self.GetWorkflowname() is not defined yet. Hence, a question mark is given instead of a value.

Trustworthiness

If a dependency is defined for a context property value then this dependency can be used to automatically extract the value. If a context property value is extracted newly each time when checking its element for compliance with a CoCon and if the (automatical) extraction mechanism works correctly then the value is correct and up-to-date. Moreover, the extraction mechanism ensures that a value is available at all. Hence, defining dependent context property values facilitates keeping the values consistent.

Example: Configurations

Different values of the same context property can be associated with the same model element in different configurations. If the same system is installed for different customers, each configuration describes the context of one installation. This can be reflected in models by using dependent context property values. Figure 4.2 illustrates how to model two different configurations 'BSH' (Building Society Schwäbisch Hall) and 'TUB' (Technical University Berlin) via the context property 'Configuration'.

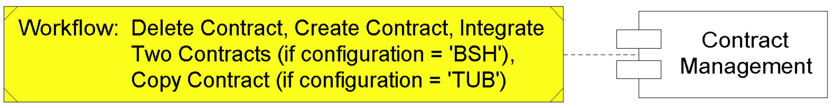
Workflow: Delete Contract, Create Contract, Integrate Two Contracts (if configuration = 'BSH'), Copy Contract (if configuration = 'TUB')
Contract Management

Figure 4.2: Context Property Values Can Depend On The Current Configuration

Dependent Context Property Values

First, the context property 'Configuration' having the valid values 'BSH' and 'TUB' is defined. Then, context-condition-dependent values of the property values 'Workflow' are associated with the model element $e =$'Contract Management' as illustrated in figure 4.2. The two values 'Delete Contract' and 'Create Contract' do not depend on the current configuration - they are *in use* for 'Contract Management' in every configuration of the system. On the contrary, the value 'Integrate Two Contracts' is a dependent context property value. It is only *in use* if the context property 'Configuration' has the value 'BSH' for the same element $e =$'Contract Management'. As well, 'Copy Contract' is a dependent value of 'Workflow'.It is only *in use* in the configuration 'TUB'. Hence, different configuration of the same model can be expressed.

4.5.3 Belongs-To Relations Result in Derived Context Properties Values

This section explains the difference between *directly associated* and *derived* context property values. A concept for deriving values is informally presented before it is defined formally.

Do We Need This?

As soon as context property values are associated with *different* elements that *belong to* each other, the mechanism introduced in this section facilitates managing the elements context property values. For example, a component *belongs to* the computer to which it is deployed. This section explains a mechanism that enables designers to associate one context property value only once to the computer in order to associate it with all components deployed to this computer automatically. This mechanism is defined independent of specific artefact types. If you prefer to associate the same context property value to each component on this computer instead of attaching it once to the computer, you can skip this section.

Def. Derivable Context Property

Not every context property is derivable. For example, the 'price' of a computer can be associated with a computer via the context property 'price'. This context property is not derivable because the values of 'price' associated with the parts may not be the same as the computer's value. Instead, the composite's price is usually the sum of all its parts. Hence, the value of 'price' of a part does not equal the price of its aggregate. On the contrary, the values of derivable context properties apply to both the computer and to all its parts. A derivable context property is defined as 2-tupel (dcp, VV^{dcp}):

1. CP is the set of the names of all context properties in the system.
2. $DCP \subseteq CP$ is the set of the names of all *derivable* context properties in the system.
3. $dcp \in DCP$ is the **name** of one derivable context property.
4. If the value(s) of the *derivable* context property dcp are associated with the element e then the element(s) that belong to e derive e's value(s) of dcp automatically due to a belongs-to relation as defined next.

Def. Belongs-To Relation

Let one element e_k belong to another element e_l with $k \neq l$. If a '**belongs-to relation** is defined between e_k and e_l then the context property values associated with e_l also apply to e_k:

1. The values of the derivable context property $dcp \in DCP$ associated with the element $e_l \in E$ are derived to another element $e_k \in E$ via the directed and transitive **belongs-to relation** $\overset{belongs}{\longrightarrow} \subseteq E \times E$. An alternative notation for $(e_k, e_l) \in \overset{belongs}{\longrightarrow}$ is: $e_k \overset{belongs}{\longrightarrow} e_l$
2. If one element $e_k \in E$ belongs to another element $e_l \in E$ via $e_k \overset{belongs}{\longrightarrow} e_l$ then e_k derives the values of the derivable context property $dcp \in DCP$ from e_l via the directed and transitive **derivedvalues-mapping** $derivedvalues_{dcp} : E \to \mathcal{P}^{VV^{dcp}}$:

$$derivedvalues_{dcp}(e) = \bigcup_{e_i : e \overset{belongs}{\longrightarrow} e_i} derivedvalues_{dcp}(e_i) \cup directvalues_{dcp}(e_i)$$

3. When referring to the context property values of one element e it won't matter if the value is directly associated with e or derived from another element to which e belongs. Hence, the directed and transitive **values-mapping** : refers to both kinds of values:

$$values_{dcp} : E \to \mathcal{P}^{VV^{dcp}}$$

$$values_{dcp}(e) = directvalues_{dcp}(e) \;\cup\; derivedvalues_{dcp}(e)$$

4. When defining a belongs-to relation then a **belongs-to criteria** must be given in natural or formal language that describes why $e_k \overset{belongs}{\longrightarrow} e_l$.

When do Element belong-to each other?

Somehow, each element of a system belongs to each other element. The only reason for defining belongs-to relations is to derive context property values, though. If a belongs-to relation is defined between all system elements then one context property value associated with one element is derived to all other elements. In such a system, CoCons cannot be applied because a context condition either selects all elements or none. Furthermore, even if two elements somehow belong to each other it is wrong to define a belongs-to relation between them if the context property values cannot be derived from one element to the other one. Therefore, this thesis focuses on only two different kinds of useful belongs-to criteria:

- A belongs-to relation exists between elements that are *part of* each other. For example, a component deployed on a computer is part of this computer. Hence, the component belongs-to the computer on which it is deployed.
- Moreover, a belongs-to relation exists between elements that *invoke* each other at runtime – during the call execution, the invoked element derives the context property values of the calling element. Modern middleware platforms, e.g. EJB or .NET support passing the context from the caller to the callee during an invocation: the called component can obtain information about the calling client via the so-called 'call context'.

Examples on refining these criteria for UML models of component-based systems are discussed in [Büb02a].

'Directly Associated' Values

Only context property values that are *'directly associated with'* an element via the $directvalues_{cp}$ mapping are called *directly associated* values. Values of a non-derivable context-property can only be directly associated.

'Derived Values'

Besides the context property values directly associated with an element e, other **derived values** are associated with e via the $derivedvalues_{cp}(e)$ mapping from every element e_i to which e belongs. When implementing a context-property-aware tool, e.g. a modelling tool, only the *directly associated* values must be made persistent because the derived values can be obtained from the associated values.

Type $\overset{belongs}{\longrightarrow}$ Type

Both types and instances of a type can exist in a system. A belongs-to relation between two types has the following impact on the instances of these types. Let both elements e_1 and e_2 be types. If $e_1 \overset{belongs}{\longrightarrow} e_2$ then any instance of e_1 **implicitely** *belong to* an instance of e_2 if these instances fit the belongs to criteria and if the criteria is automatically decidable.

Explicit Belongs-To Relations

A metamodel contains *types* of model elements. The instance of a metamodel element is a model element. Hence, *implicit* belongs-to relations for UML diagrams are defined by specifying explicit belongs-to relations between elements of the UML *metamodel* as demonstrated in [Büb02a]. The benefit of only explicitly specifying belongs-to relations in the metamodel is that during design or in later abstraction levels, only implicit belongs-to relations are used. This is less confusing.

Benefit: Belongs-To Hierarchy

The consequence of defining a belongs-to relation is that a context property value associated with one element applies to others, too. If this value changes, it must not be modified at every element involved. Instead, it only must be modified at one element, and the change is automatically propagated to the other elements that belong to this element. Moreover, belongs-to relations create a hierarchy of context property values because they are transitive: if $a \overset{belongs}{\longrightarrow} b \overset{belongs}{\longrightarrow} c$ then $a \overset{belongs}{\longrightarrow} c$. Thus, a context property value associated with c automatically is associated with b and a. This belongs-to hierarchy provides an useful structure. It enables the designer to associate a context property value with the element that is as high as possible in the belongs-to hierarchy. It must be *directly associated* only once and, thus, is *derived to* probably many elements. Hence, redundant and possibly inconsistent context property values can be avoided, and the comprehensibility is increased. Belongs-To relations can be used for automatically deriving context property values from other system or model elements. Hence, they facilitate preventing wrong context property values.

Example

As discussed in section 3.2.4, inter-value constraints can prevent illegal context property values. For example: an inter-value constraint can forbid that the context property 'Personal Data' can have both the value 'True' *and* 'False' for the same element. Of course, derived values must not violate inter-value constraints, too. Let's assume that a component *installed in* a container *belongs to* this container: *Components* $\xrightarrow{belongs}$ *Container*. This belongs-to relation is expressed on type level - it implicitely applies to all component instances and their container instances. According to this belongs-to relation, *all* components deployed in a container do not handle personal data if the value 'False' of the context property 'Personal Data' is associated with their container. If any of these component additionally has the value 'True' of the context property 'Personal Data' then it violates the inter-value constraint stating that no element can have both values 'True' and 'False' in the same time. However, how can it be allowed to have both kinds of components in the same container – those who handle personal data, and those who don't? If some components in the container handle personal data and some others don't, neither 'True' nor 'False' can be associated with the container because each value is in conflict with some of the components in the container. This example shall demonstrate the benefits of a belongs-to hierarchy. Due to automatically derived values, illegal associations of values with elements can be detected. Associating 'False' to a container filled with both 'False' *and* 'True' components simply is bad design. If a value is associated with an element then it applies to *all* elements that belong to this element via a belongs-to relation. By associating 'False' to the container, no component having 'True' is allowed in this container. By associating no value to the container at all, both 'True' and 'False' are allowed values for components in this container. Hence, inconsistent context property values (bad design) can be avoided via belongs-to relations.

4.5.4 Outlook: Applying Context Properties in Continuous Software Engineering

Change Estimation

Context properties allow *subject-specific, problem-oriented views* to be concentrated on. If the context is known in which the modification takes place and if the system's elements are annotated via a corresponding context property then this context property facilitates to identify those elements that are likely to be involved in the modification. For instance, only those elements belonging to workflow 'X' may be of interest when modifying the workflow 'X'. Of course, other elements not involved in the workflow 'X' may be involved in the change due to dependencies as explained in section 4.1. Moreover, It may be necessary to modify only a few of the elements needed in the workflow 'X' when changing the workflow 'X'. Thus, taking only the context property values of the elements into account does not result in a precise change impact analysis. Instead, it assists in estimating the change impact at first glance. If only a few elements are needed in the workflow 'X' according to their context property values then the change impact might be less compared to a modification where many elements belong the context in which the modification takes place.

Context can Classify Changes

According to [Par94], predicting changes is about as difficult as predicting future. Still he thinks we could classify different kinds of change and then assign a certain probability for each of these change types. We would then have to consider in advance at least the more probable changes. Context properties can assist in classifying different kinds of changes by estimating how probably a change will happen within each context is. For instance, certain workflows might be bound to change frequently, while others haven't changed for years. By first annotating context property values to the elements and then estimating change probability of the context property values an architect can identify those elements that are more likely to change than the others. However, this thesis does not discuss probability models based on context in detail because the focus of the thesis is on defining invariants. The context properties themselves are not considered as invariants. Instead, the context-based constraints that refer to these context properties are discussed here in detail. Based on an understanding of the expected future evolution of the software system, the software architect can employ various design solutions to prepare for the future incorporation of new and changed requirements. Context properties facilitate identifying the hot spots of possible future evolution.

5. The Context-Based Constraint Language CCL

5.1 Overview on CCL

This chapter introduces the **C***ontext-based* **C***onstraint* **L***anguage CCL*. CCL has 22 CoCon-predicates for expressing requirements for component-based systems. Only the artefact-type-*independent* semantics of these 22 CoCon-predicates are discussed in this chapter. The artefact-type-specific semantics of these 22 CoCon-predicates for UML models will be discussed in chapter 6. Future research will probably identify additional CoCon-predicates. Chapter 7 will present a method for identifying those CCL CoCon-predicates that are relevant for a specific application. This method will explicitly address adding new CoCon-predicates to CCL in section 7.2.1.

CoCon Family A **CoCon family** groups CoCon-predicates with related semantics. In each of the oncoming sections, CoCon-predicates of one CoCon family are introduced.

Focus: Components A system artefact can contain different element types. Most examples given in the next sections only refer to one element type: 'components'.

`(NOT | ONLY)` Each CoCon-predicate can combined with the CoCon-predicate operation 'NOT' or 'ONLY' after the keyword MUST. For example, the CoCon-predicate `ACCESSIBLE TO` can either state that certain elements `MUST BE ACCESSIBLE TO` other elements, or that they `MUST NOT BE ACCESSIBLE TO` other elements, or that they `MUST ONLY BE ACCESSIBLE TO` other elements. The abbreviation '`(NOT | ONLY)`' is used to refer to all three possible CoCon-predicate operations of one CoCon-predicate in the next sections.

5.2 Access Permission CoCons

5.2.1 The Notion of Access Permission CoCons

Goal: Security CoCon-predicates of the accessibility family determine if and how elements can access other elements. Thus, they facilitate handling security requirements.

Constrained Element Types The target set of an access permission CoCon can contain any element type that can access other elements, such as 'components'. As well, the scope set of access permission CoCons can contain any element type that can be accessed by other elements. However, nothing but 'components' in the target sets and scope sets of access permission CoCons are discussed here.

5.2.2 Access Permission CoCon-predicates

`ACCESSIBLE TO` CoCons A `(NOT | ONLY)` **ACCESSIBLE TO** CoCon defines that the components in its target set are `(NOT | ONLY)` accessible to the components in the scope set.

`READABLE BY` CoCons A `(NOT | ONLY)` **READABLE BY** CoCon defines that the components in its target set are (NOT | ONLY) readable by any of the components in the scope set.

`WRITEABLE BY` CoCons A (NOT | ONLY) **WRITEABLE BY** CoCon defines that all the components in its scope set are (NOT | ONLY) writeable by any of the components in its target set.

`EXECUTABLE BY` CoCons A (NOT | ONLY) **EXECUTABLE BY** CoCon defines that all the components in its scope set must (NOT | ONLY) be able to invoke the operations of any of the components in its target set.

`REMOVEABLE BY` CoCons A (NOT | ONLY) **REMOVEABLE BY** CoCon defines that all the elements in its scope set must (NOT | ONLY) be removed by any of the elements in its target set.

5.2.3 Detectable Inter-CoCon Conflicts of Access Permission CoCons

This sections discusses the detection of contradicting access permission CoCons. The general inter-CoCon conflicts for CoCons having the*same* CoCon-predicate presented in section 4.3.1 apply to each access permission CoCon-predicate. This section discusses additional inter-CoCon conflicts for access permission CoCons of *different* CoCon-predicates.

Each of the CoCon-predicates `READABLE BY`, `WRITEABLE BY`, `EXECUTEABLE BY`, and `REMOVABLE BY` is a refinement of the CoCon-predicate `ACCESSIBLE TO`. For example, any element e_i that is readably by another element e_k also is accessible to e_k. Therefore, the four generic inter-CoCon conflicts of section can be refined as follows: the generic inter-CoCon conflicts also apply if one of the two CoCons $C_1(x, y)$ or $C_2(x, y)$ is an `ACCESSIBLE TO` CoCon while the other one is a `READABLE BY`, a `WRITEABLE BY`, an `EXECUTEABLE BY`, or a `REMOVABLE BY` CoCon. For example, the refined generic inter-CoCon conflicts 1 states that no element e_i can both be `ACCESSIBLE TO` and `NOT READABLE BY` any e_k.

5.2.4 Example for Using Access Permission CoCons

Privacy Policy The privacy policy example given in section 2.2 is discussed throughout the thesis. It can be specified in CCL as:

```
ALL COMPONENTS WHERE 'Personal Data' EQUALS 'True'
MUST NOT BE ACCESSIBLE TO
ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Create Report'
```

Reflexive CoCons If the same component has both the value 'Yes' for its context property 'Personal Data' *and* the value 'Create Report' for its context property 'Workflow' then it belongs *both* to the target set and to the scope set of the privacy policy CoCon. This is absurd, of course. It means that this component cannot access itself. Such bad design is detected via the 'NOT-SELF' inter-CoCon conflict defined in section 4.3.1.

The 'NOT-SELF' inter-CoCon conflict applies because access permission CoCons are reflexive ($\forall x : C(x, x)$). Every component involved in this conflict must be changed until it *either* handles personal data *or* belongs to the 'Create Report' workflow. It may not belong to *both* contexts. If it is not possible to adjust the components accordingly then they cannot comply with this requirement.

5.2.5 Related Research on Access Control Policies

Access Control Policies According to [SdV01], a **access control policy** is concerned with permitting only authorised users (**subjects**) to access services and resources (**targets**). Many different approaches for expressing access control policies exist. Typically, the *subjects* can indirectly be selected via their role. A role is a grouping mechanism similar to context properties: a **role** permits the grouping of a set of permissions related to a position in an organisation such as finance director or physician. It allows permissions to be defined in terms of the position rather than the person assigned to the permission. Indirect selection of subjects according to their role has a long tradition in access control policies. CoCons enhance this notion. They can regard the role of an element or a user as context and, thus, select the constrained elements or users via their role. Additionally, they can select elements or users according to their current context, e.g. their current location, their hobby or their current workflow.

Access Permission CoCons can both indirectly select the subjects *and* targets according to their current context. On the contrary, most access control policies only consider the indirect selection of their subjects (the users) via roles. One approach in which the *targets* can indirectly be selected via roles is the policy core information model (PCIM) presented in [MESW01]. In PCIM, a role represents a functional characteristic or capability of a resource to which policies are applied, such as backbone interface, frame relay interface, web-server, firewall, etc. Roles are used as a mechanism for indirectly associating policies with the network elements to which the policies apply. However, PCIM roles only refer to *internal context*, while CoCons also refer to external context as explained in section 3.2.1.

Ponder

Another recent approach can indirectly select the constrained targets: the Ponder language for specifying management and security policies is defined in [Dam02]. 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 2.2 can be defined in Ponder as:

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

All in all, Ponder could be used instead of the context-based constraint language CCL defined in [Büb02a] in order to express CoCons. Still, Ponder differs from CoCons in several ways. CoCons neither consider events nor actions, while Ponder has operational semantics. Furthermore, Ponder uses *domains* (see section 3.2.6) in order to select the object to which a policies applies, while CoCons identify their constrained elements according to their context properties. While a context property is a typed attributes having a name and value(s), a domain is a flat string with hierarchy. Moreover, Ponder cannot directly select the constrained elements, while CoCons can. And finally, Ponder has no two-step semantics definition. Ponder misses this additional abstraction layer that allows to handle requirements for different artefact types.

5.3 Communication CoCons

Communication CoCons describe how to handle communication calls. Therefore, they are typically monitored and applied at runtime or during configuration.

5.3.1 The Notion of Communication CoCons

Goal: Intercepting Communication Calls

If the context property values of the components involved in one communication call, such as a remote procedure call or an asynchronous message, fit to the context condition of a communication CoCon then a **context-aware service** is invoked that checks whether the current call complies with the communication CoCon. Details are described in [LBBK03, Bil02, Wan02]. The context-aware services suggested here handle non-functional requirements. They don't modify the call's content.

Constrained Element Types

The target set and the scope set of a communication CoCon can contain any element type that can communicate with other elements. However, nothing but 'components' in the target set or the scope set of communication CoCons are discussed here.

5.3.2 The Communication CoCon-predicates

The family of communication CoCons consists of several CoCon-predicates:

`CACHED WHEN CALLING`

A (NOT | ONLY) **CACHED WHEN CALLING** CoCon specifies that (ONLY) a communication call from a component in its target set to a component in its scope set must (NOT) be cached.

`ENCRYPTED WHEN CALLING`

A (NOT | ONLY) **ENCRYPTED WHEN CALLING** CoCon specifies that (ONLY) a communication call from a component in its target set to a component in its scope set must (NOT) be encrypted.

`ERRORHANDLED WHEN CALLING`

A (NOT | ONLY) **ERRORHANDLED WHEN CALLING** CoCon specifies that (ONLY) a communication call from a component in its target set to a component in its scope set must (NOT) be error-handled.

`LOGGED WHEN CALLING`

A (NOT | ONLY) **LOGGED WHEN CALLING** CoCon specifies that (ONLY) a communication call from a component in its target set to a component in its scope set must (NOT) be logged.

The following communication CoCon-predicates are typically applied during configuration:

`PROTECTED BY A TRANSACTION WHEN CALLING`

A (NOT | ONLY) **PROTECTED BY A TRANSACTION WHEN CALLING** CoCon specifies that (ONLY) a communication call from a component in its target set to a component in its scope set must (NOT) be protected by a transaction mechanism.

`SYNCHRONOUSLY CALLING`

A (NOT | ONLY) **SYNCHRONOUSLY CALLING** CoCon specifies that (ONLY) the elements of the target set must (NOT) synchronously invoke the elements in the scope set.

`ASYNCHRONOUSLY CALLING`

A (NOT | ONLY) **ASYNCHRONOUSLY CALLING** CoCon specifies that (ONLY) the elements of the target set must (NOT) asynchronously invoke the elements in the scope set.

5.3.3 Detectable Inter-CoCon Conflicts of Communication CoCons

This sections discusses the detection of contradicting communication CoCons. In addition to the general inter-CoCon conflicts for CoCons having the*same* CoCon-predicate presented in section 4.3.1, this section introduces inter-CoCon conflicts for communication CoCons of *different* CoCon-predicates. The elements e_i and e_k are target or scope set elements of communication CoCons with $i \neq k$. An inter-CoCon conflict exists if any of the following inter-CoCon conflicts is violated:

1. No element e_i can both be `SYNCHRONOUSLY CALLING` and `ASYNCHRONOUSLY CALLING` any e_k

5.3.4 Examples for Using Communication CoCons

Logging

The requirement "every remote procedure call (RPC) belonging to the workflow 'Integrate Two Contracts' must be recorded in a log-file" can be specified via the following communication CoCon:

```
ALL COMPONENTS MUST BE LOGGED WHEN CALLING ALL COMPONENTS WHERE
'Workflow' CONTAINS 'Integrate Two Contracts'.
```

This communication CoCon defines that a context-aware logging-service is invoked if a communication call belongs to the workflow 'Integrate Two Contracts'.

5.3.5 Related Research on Communication CoCons

Some of the communication CoCon-predicates reflect issues addressed in policy-based resource management. However, most of the policy-based management approaches use (event-)condition-action (ECA) rules. As explained in section 3.4, CoCons don't have operational semantics. They can be turned into ECA rules by adding events and actions as described in section 3.4. This thesis mostly ignores events and actions, though.

Path-Based Policy Language

Network policies define the relationship between clients using network resources and the network elements that provide those resources. The main interest in network policies is to manage and control the quality of service (QoS) experienced by networked applications and users, by configuring network elements using policy rules. For example, the Internet Engineering Task Force (IETF) policy model defined in [MESW01] considers policies as rules that specify actions to be performed in response to defined conditions: `if <condition(s)> then <action(s)>`. There is no explicit event specification to trigger the execution of the actions. Instead, the IETF policy model assumes that an implicit event such as a particular traffic flow, or a user request will trigger the policy rule. Other approaches to network policy specification try to extend the IETF rule-based approach to specify traffic control using a concrete language. An example is the path-based policy language (PPL) described in [SLX01]. The language is based on the idea of providing better control over the traffic in a network by constraining the path the traffic must take. However, PPL expressions cannot indirectly select the constrained components as CoCons can. Communication CoCons can select the constrained communication path according to the current context of the caller or the callee. Therefore, a CoCon can apply to new communication paths if the context of the caller or the callee changes. This adaptive approach is not considered in PPL.

5.4 Distribution CoCons

5.4.1 The Notion of Distribution CoCons

Goal: Designing Distribution

Distribution CoCons determine whether the target components have to be available at the CoCon's scope elements or not. The are introduced in [BL01]. A revised version has been published in [Büb03].

Constrained Element Types

The target set of a distribution CoCon can contain any element type that can be contained in other elements, such as 'components' can be contained in 'containers'. As well, the scope set of distribution CoCons can contain any element type that can contain the other element type of the target set. However, nothing but 'components' in the target sets and 'containers' or 'computers' in the scope sets of distribution CoCons are discussed here.

Availability

Distribution CoCons facilitate designing distributed systems as follows. 'availability' and 'load balance' are contradicting distribution goals. Availability is optimal if every element is allocated to every computer because each computer can still access each element even if the network or other computers of the system have crashed. However, this optimal availability causes bad load balance because each modification of an element must be replicated to every other computer of the system. Typically, the limits of hardware performance and network bandwidth don't allow optimal availability. Instead, a reasonable trade of between availability and load balance must be achieved by clustering *related* data. Those elements that are related should be allocated to the computers where they are needed in order to improve their availability within this cluster.

5.4.2 Distribution CoCon-predicates

`ALLOCATED TO` CoCons

A `(NOT | ONLY) ALLOCATED TO` CoCon defines that the components in its target set must `(NOT | ONLY)` be deployed on the containers or the computers in its scope set.

Replication is well known in distributed databases. As, e.g., explained in [Har02], replication can be realised with current middleware platforms, too. In this thesis, the term 'a component is replicated' means that the component's state is serialized and the resulting data is copied. The following CoCon-predicates handle replication.

`SYNCHRONOUSLY REPLICATED TO` CoCons

A `(NOT | ONLY)` **SYNCHRONOUSLY REPLICATED TO** CoCon defines that the components in its target set must `(NOT | ONLY)` be synchronously replicated from where they are allocated to – specified via `ALLOCATED TO` CoCons – to the elements in its scope set.

`ASYNCHRONOUSLY REPLICATED TO` CoCons

A `(NOT | ONLY)` **ASYNCHRONOUSLY REPLICATED TO** CoCon defines that the components in its target set must `(NOT | ONLY)` be asynchronously replicated from their allocation – their allocation is specified via `ALLOCATED TO` CoCons – to the elements in its scope set.

5.4.3 Detectable Inter-CoCon Conflicts of Distribution CoCons

This sections discusses the detection of contradicting distribution CoCons. In addition to the general inter-CoCon conflicts for CoCons having the*same* CoCon-predicate presented in section 4.3.1, this section introduces inter-CoCon conflicts for distribution CoCons of *different* CoCon-predicates. The elements e_i and e_k are target or scope set elements of distribution CoCons with $i \neq k$. An inter-CoCon conflict exists if any of the following inter-CoCon conflicts is violated:

1. No element e_i may be both `NOT ALLOCATED TO` e_k and `SYNCHRONOUSLY REPLICATED TO` e_k.
2. No element e_i may be both `NOT ALLOCATED TO` e_k and `ASYNCHRONOUSLY REPLICATED TO` e_k.
3. No element e_i may be both `ALLOCATED TO` e_k and `SYNCHRONOUSLY REPLICATED TO` e_k.

4. No element e_i may be both `ALLOCATED TO` e_k and `ASYNCHRONOUSLY REPLICATED TO` e_k.
5. No element e_i may be both `SYNCHRONOUSLY REPLICATED TO` e_k and `ASYNCHRONOUSLY REPLICATED TO` e_k.

5.4.4 Examples for Using Distribution CoCons

'Availability' Example

Take, for instance, an 'availability' requirement stating that *"all components needed by the workflow 'NewCustomer' must be allocated to the computer 'Mainframe' in order to be able to execute this workflow on this computer even if the network connection fails"*. This 'availability' requirement can be written down via CCL as follows: `ALL COMPONENTS WHERE 'Workflow' CONTAINS 'NewCustomer' MUST BE ALLOCATED TO THE COMPUTER 'Mainframe'`.

5.4.5 Related Research on Distribution and Network Policies

ADLs

One way in which we cope with large and complex systems is to abstract away some of the detail, considering them at an architectural level as composition of interacting objects. To this end, the variously termed *Coordination, Configuration and Architectural Description Languages* facilitate description, comprehension and reasoning at that level, providing a clean separations of concerns and facilitating reuse. According to [KM97], in the search to provide sufficient detail for reasoning, analysis or construction, many approaches are in danger of obscuring the essential structural aspect of the architecture, thereby losing the benefit of abstraction. On the contrary, CoCons stay on an abstract level in order to keep my approach simple.

Context-specific Clusters

Aspect-oriented languages supplement programming languages with properties that address design decisions. According to [KLM$^+$97], these properties are called *aspects* and are incorporated into the source code. Most aspect-oriented languages do not deal with expressing design decisions in during *design*. D^2AL ([Bec98]) differs from the other aspect oriented languages in that it is based on the system model, not on its implementation. Objects that interact heavily must be located together. 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 [RE96] that, likewise, expresses the architecture explicit 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 allocate objects together because of shared context instead of direct collaboration. They assist in grouping related objects into **context-specific clusters** and define how to allocate or replicate the whole cluster.

After distributed applications became popular and sophisticated in the 80s, over 100 *programming languages* specifically for *implementing* distributed applications were invented according to [BST89]. Nevertheless, hardly anyone took distribution into consideration already on the *design* level. Up to now, the rationale for distribution decisions is barely recorded during the design and development of software system. A distribution decision is typically taken into account during implementation and is expressed directly in configuration files or in source code. But, when adapting a system to new, altered, or deleted requirements, existing distribution decisions should not unintentionally be violated. By expressing them in an implementation-independent way they can be considered at different abstraction levels throughout the lifetime of a distributed system.

5.5 Information-Need CoCons

The CoCon families presented in the previous sections all describe non-functional requirements. CoCons are not restricted to expressing non-functional requirements, though. This section will demonstrate that CoCons also can express functional requirements. Therefore, it ends with an additional subsection that discusses appropriate events and actions.

5.5.1 The Notion of Information-Need CoCons

Goal: Information Supply

The central problem of information supply today is no longer quantity but quality. The individuals are only interested in a tiny fraction of the available data. In order to deliver the *right* information to a user *when* it is required and *where* it is required, the *information need* of the user must be defined. This section suggests to declaratively define *information need* via information-need CoCons. They can specify which users are (or are not) interested in which documents depending on the current context of the users or documents.

Information Need CoCons

The information need of users can change according to the current context of the users or the documents. In order to provide the *right* information to the user *where* needed and *when* it is required, the definition of 'who must be notified of what' should consider the *current context* of users and documents. Information-Need CoCons specify which elements are interested in which elements depending on the current context of the elements. Moreover, they specify with elements are as interesting as other elements.

5.5.2 The Information-Need CoCon-predicates

One user can be interested in several documents possibly not stored in the same place, repository, or directory. Moreover, one document can be interesting for several possibly unrelated users. An information-need CoCon can indirectly select the constrained users and documents via their metadata. The family of information-need CoCons consists of the several CoCon-predicates.

`NOTIFIED OF` CoCons

A `(NOT | ONLY) NOTIFIED OF` CoCon specifies that the users in its target set are `(NOT | ONLY)` interested in the documents in its scope set – the target set users must `(NOT | ONLY)` be notified of the scope set documents.

`AS INTERESTING AS` CoCons

A `(NOT | ONLY) AS INTERESTING AS` CoCon specifies that any document in its target set is `(NOT | ONLY)` as interesting as any scope set document. It is bi-directional (symmetric)– whoever is interested in its target set documents is also (| `NOT` | `ONLY`) interested in all its scope set documents and vice versa.

`AVAILABLE TO ANYONE INTERESTED IN` CoCons

A `(NOT | ONLY) AVAILABLE TO ANYONE INTERESTED IN` CoCon specifies that any document in its target set must be (NOT | ONLY) be available to anyone interested in any scope set document. In contrast to AS INTERESTING AS CoCons, the AVAILABLE TO ANYONE INTERESTED IN CoCons are unidirectional – whoever is interested in its scope set document(s) is also | (NOT | ONLY) interested in all its target set documents, but not vice versa.

AS INTERESTED AS CoCons

A (NOT | ONLY) AS INTERESTED AS CoCon specifies that any user in its target set is (NOT | ONLY) interested in the same documents as any user in the scope set. It is bi-directional (symmetric).

NOTIFIED OF THE SAME AS CoCons

A (NOT | ONLY) NOTIFIED OF THE SAME DOCUMENTS AS CoCon specifies that any user in its target set is (NOT | ONLY) interested in the same documents as any user in the scope set. In contrast to AS INTERESTED AS CoCons, the NOTIFIED OF THE SAME DOCUMENTS AS CoCons are unidirectional – they do *not* define that the users in their scope sets are interested in the same documents as the target set users.

Ignoring Either Users or Documents

An NOTIFIED OF CoCon defines *who* is (not) interested in *what*. Its target set contains the interested users (*who*). Nevertheless, if it doesn't matter *who* is interested then it can be ignored when expressing information need. Only NOTIFIED OF CoCons define *who* is (not) interested in *what*. The other CoCon-predicates discussed here express simplified information need:

- Both AS INTERESTING AS CoCons and AVAILABLE TO ANYONE INTERESTED IN CoCons do not define *who* is interested in the documents in their target or scope sets. If no NOTIFIED OF CoCon defines *who* is interested in these documents then simply all users of the system must be notified.
- Both AS INTERESTED AS CoCons and NOTIFIED OF THE SAME AS CoCons do not define *what* documents are interesting for the users in their target or scope set. If no NOTIFIED OF CoCon defines in *what* documents these users are interested then these users must simply be NOTIFIED OF all documents managed by the system.

I recommend focusing on NOTIFIED OF CoCons when specifying information need. The other CoCon-predicates should only be used to extend information need defined via NOTIFIED OF CoCons.

Constrained Element Types

The target set of an NOTIFIED OF CoCon can contain any element type that can be interested in information. Typically, the target set contains users or components. The scope set of an information-need CoCon contains any element type in which someone or something can be interested, like documents or business types. In case of an AS INTERESTING AS CoCon or an AVAILABLE TO ANYONE INTERESTED IN CoCon, the target set and the scope set must contain elements of only one type. This element type usually is a document.

5.5.3 Detectable Inter-CoCon Conflicts of Information-Need CoCons

This sections discusses the detection of contradicting information-need CoCons. In addition to the general inter-CoCon conflicts for CoCons having the*same* CoCon-predicate presented in section 4.3.1, this section introduces inter-CoCon conflicts for information-need CoCons of *different* CoCon-predicates. The elements e_i and e_k are target or scope set elements of information-need CoCons with $i \neq k$. An inter-CoCon conflict exists if any of the following inter-CoCon conflicts is violated:

1. If the element e_i must be `NOTIFIED OF` e_k and e_k is `AS INTERESTING AS` e_m then e_i must be `NOTIFIED OF` e_m, too.
2. If the element e_i must be `NOT NOTIFIED OF` e_k and e_k is `AS INTERESTING AS` e_m then e_i must be `NOT NOTIFIED OF` e_m, too.
3. If the element e_i must be `NOTIFIED OF` e_k and e_m is `NOT AS INTERESTING AS` e_k then e_i must not be `NOTIFIED OF` e_m.
4. If the element e_i must be `NOTIFIED OF` e_k and e_m is `AVAILABLE TO ANYONE INTERESTED IN` e_k then e_i must be `NOTIFIED OF` e_m, too.
5. If the element e_i must be `NOT NOTIFIED OF` e_k and e_k is `AVAILABLE TO ANYONE INTERESTED IN` e_m then e_i must be `NOT NOTIFIED OF` e_m, too.
6. If the element e_i must be `NOTIFIED OF` e_k and e_m is `NOT AVAILABLE TO ANYONE INTERESTED IN` e_k then e_i must not be `NOTIFIED OF` e_m.
7. If the element e_i must be `AS INTERESTING AS` e_k then e_i must not be `NOT AVAILABLE TO ANYONE INTERESTED IN` e_k
8. If the element e_i must be `NOTIFIED OF` e_k and e_m is `AS INTERESTED AS` e_i then e_m must be `NOTIFIED OF` e_k, too.
9. If the element e_i must be `NOT NOTIFIED OF` e_k and e_i is `AS INTERESTED AS` e_m then e_m must be `NOT NOTIFIED OF` e_k, too.
10. If the element e_i must be `NOTIFIED OF` e_k and e_m is `NOT AS INTERESTED AS` e_i then e_m must not be `NOTIFIED OF` e_k.
11. If the element e_i must be `NOTIFIED OF` e_k and e_m is `NOTIFIED OF THE SAME DOCUMENTS AS` e_i then e_m must be `NOTIFIED OF` e_k, too.
12. If the element e_i must be `NOT NOTIFIED OF` e_k and e_k is `NOTIFIED OF THE SAME DOCUMENTS AS` e_m then e_i must be `NOT NOTIFIED OF` e_m, too.
13. If the element e_i must be `NOTIFIED OF` e_k and e_m is `NOT NOTIFIED OF THE SAME DOCUMENTS AS` e_k then e_i must not be `NOTIFIED OF` e_m.
14. If the element e_i must be `AS INTERESTED AS` e_k then e_i must not be `NOT NOTIFIED OF THE SAME DOCUMENTS AS` e_k

5.5.4 Examples for Using Information-Need CoCons

E-Learning

All students who attend a lecture should read those books that fit to the topic of the lecture:

```
ALL USERS WHERE ‘Role’ CONTAINS ‘Student’
MUST BE NOTIFIED OF
ALL DOCUMENTS WHERE ‘Topic’ INTERSECTS WITH ‘User.Lecturetopic’
```

Dot-Path-Notation

In this example, the Dot-Path-notation explained in section 3.3.4 is used in the scope set context condition in order to refer to the ‘other’ constrained element. A document is selected by this context condition if at least one of its associated values of the context property ‘Topic’ equals the ‘Lecturetopic’ values associated with the user. A CoCon defines a relation between any element of its target set and any element of its scope set. Usually, the context condition for selecting the elements of one of these sets only checks the values associated with the currently checked element. In this example, however, the prefix 'User' defines that this scope set context condition does not refer to the document's context property values. Instead, it refers to the context property values associated with the related element of the 'other' set - the user.

Mobile Devices

Mobile services and mobile applications should reflect the current context. In this example, the current location of the mobile device and the user profile of the person who uses the device are considered via an information-need CoCon:

```
ALL USERS MUST NOT BE NOTIFIED OF ALL DOCUMENTS WHERE
'DOCUMENT.DescribedCity' DOES NOT EQUAL 'USER.Location.City' OR
'DOCUMENT.Genre' DOES NOT INTERSECT WITH 'USER.Hobby'.
```

It filters the *right* information out of a flood of data. Documents are ignored if they don't match with the user's hobbies or don't apply to the city in which the user currently is located.

Navigation

In this example, the Dot-Path-notation is used in the scope set context condition in order to navigate. As explained in section 3.3.4, navigation is only possible if an association between the elements is defined. The scope set context condition checks the values of 'City' associated with the user in the role of 'Location'.

5.5.5 Related Research on Information-Need CoCons

User Models

How do we know who needs to be notified of what? This question can be answered by listing the interesting documents for each user in a so called *user model* ([McT93]). User models are typically based on profiling user interests in order to distribute messages to interested users. According to [SKK01], user-adaptive systems typically learn about individual users by processing observations about individual behaviour. However, it may take a significant amount of time and a large number of observations to construct a reliable model of user interests, preferences and other characteristics. Additionally, it is useful to take advantage of the behaviour of similar users as in the collaborative filtering approach ([KMM$^+$97]) or recommender systems ([RV97]). Many other approaches exist to extract user's interest from the content of the visited web pages or documents in order to recommend other pages or documents that are relevant to her current interests. On the contrary, this section presents a mechanism for declaratively defining who must (not) be notified of what. This declarative approach cannot replace learning from user behaviour. Instead, information-need CoCons enable us to combine automatically extracted and intellectually defined knowledge about the information need of users.

Conceptual Clusters

Building user models is difficult, though, because it is not easy for users to specify what they are interested in according to [SK92]. As a typical approach for user modelling, *conceptual clusters* ([Mun97]) classify documents based on the terms they *contain* and queries are processed based on the terms specified by the user. One of the major obstacles is the vocabulary problem described in [CSN$^+$96]: The terms contained in the documents available are different from those the user would use to specify his or her interests. Information-Need CoCons, however, do not necessarily refer to the *content* of the interesting document. Instead, they can refer to its *context*. The context property values of a document can use terms found in the document, but they also can describe the context in which the document is used even if this context is not mentioned in the document at all. For instance, they can describe in which workflows the document is used. Moreover, they can describe the document's context (or content) in terms used by the user.

Organizational Concept Space

Recent approaches, like the *organizational concept space* ([ZKS00]), also partly consider context by incorporating *organizational information* in the user model. An interest matrix is used with documents as one dimension and user as the other. The entries of the matrix indicate the level of interest of the *individual* user in the *individual* document. On the contrary, information-need CoCons relate *groups* of users to *groups* of documents. They adapt more easily to new or changed users, documents or context because every new or changed element involved can automatically be selected according to its context. Information-Need CoCons can be combined with the interest matrix as follows: the entries of the matrix must correspond to the CoCon(s). If an information-need CoCon identifies that a certain user must (not) be notified about a certain document then the corresponding entry in the matrix can be checked for compliance with the CoCon. An entry in the matrix can contradict a CoCon. For instance, user_15 might be interested in document_52 according to the matrix, while the CoCon states that this user must not be notified of this document. In case of such a conflict, either the matrix or the CoCon should be adapted. The matrix is adapted by changing those entries that contradict the CoCon. This mechanism can assist in filling the matrix with entries at all, because one CoCon can constrain many users or documents and, thus, can set the value of many entries in the matrix.

One limitation of learning from user behaviour is that the most popular document may not be interesting for every individual. In addition, a document that is very interesting for one user may be boring for all other users. An automatically extracted user model only considers the ant trails where many users share an interest. An information need CoCon-Rule, on the contrary, can also take unconventional information-need into account.

Up to now, no policy language for expressing information need exists. One reason for it is the huge number of rules needed if every *individual* document and *individual* user shall be considered. In contrast, one CoCon-Rule can express information-need for possibly large sets of users and documents. Future research could examine how to combine declarative specification of information need policies with user models build on observation of individual user's behaviour. For example, the action-clause of a corresponding CoCon-Rule can define to set the interest matrix entry to a certain value if it does not comply with the CoCon.

5.5.6 Information-Need CoCon-Rules

The previous sections have showed how to define information need via constraints. Identifying information need is one thing, and satisfying the need is another. Nevertheless, constraints do not tell what happens when information need exists according to the constraint as discussed in section 3.4. This section outlines the consideration of events and actions in information-need CoCon rules.

The aim of information logistics is to provide the *right* information *when* it is required and *where* it is required. The *right* information is described in the scope set specification of an information-need CoCon. It can be defined via a context condition that selects the *right* elements according to their current context.

The question *when* the information is required can be answered in four ways. First, the corresponding CoCon-Rule's `COMPLIANCE-ACTION` attribute can describe when and how to deliver the information, for instance, it can demand to 'deliver the information immediately via SMS'. Second, an event that triggers the information need can be specified. Third, the user's context can cause information need. For example, users `WHERE 'Local Time' = '08:15'` can be specified to be interested. Thus, the users can be selected via a context condition that describes *when* information need exists. Fourth, the time of interest can depend on the interesting documents. Hence, a context condition can refer to the time context of an interesting document, like `DOCUMENTS WHERE 'Publication Year' = '2002'`.

Likewise, the question *where* the information is required can be specified in a context condition that describes *where* the interested user requires information or *where* the interesting documents reside. In addition, the attribute `COMPLIANCE-ACTION` can specify where the information must be delivered, and the event clause can define *at specific places* at which the users and documents must be checked for whether which CoCon applies to them.

5.6 Inter-Value CoCons

Inter-Value CoCons specify dependencies between context property values of an element.

5.6.1 The Notion of Inter-Value CoCons

Goal: Dependencies between Context property values

Inter-Value CoCons express inter-value constraints (see section 3.2.4). They define dependencies between the context property values of an element and, therefore, facilitate keeping the values consistent. Inter-Value CoCons specify that an element having certain context property values must or must not have certain other context property values.

5.6.2 The Inter-Value CoCon-predicates

The family of inter-value CoCons consists of the following CoCon-predicates.

`THE SAME AS` CoCons

The target set of a (NOT) **THE SAME AS** CoCon must (NOT) contain the same elements as its scope set. It is bi-directional (symmetric): the target set elements must (NOT) fulfil the scope set's context condition and the scope set elements must (NOT) fulfil the target set's context condition.

`FULFILLING THE CONTEXT CONDITION OF` CoCons

This is an uni-directional inter-value CoCon. For instance, all books where the 'author'= 'J.K. Rowling' must have the 'genre' = 'fantasy', but not all books of the of the 'genre' = 'fantasy' are written by the 'author'= 'J.K. Rowling'. The elements of the target set of a (NOT) **FULFILLING THE CONTEXT CONDITION OF** CoCon must (NOT) fulfil the scope set's context condition. However, the scope set elements are not matched with the context condition of the target set as with `THE SAME AS` CoCons.

`FULFIL THE CONTEXT CONDITION` CoCons

The syntax of an `FULFILLING THE CONTEXT CONDITION OF` CoCon contains an superfluous part. In the following example, the superfluous parts is printed in *ITALICS*: `ALL COMPONENTS WHERE X = 3 MUST` *BE* `FULFILLING THE CONTEXT CONDITION` *OF ALL COMPONENTS WHERE*

`Y = 2` can be abbreviated by removing the superfluous parts. The abbreviated version reads `ALL COMPONENTS WHERE X = 3 MUST FULFIL THE CONTEXT CONDITION Y = 2`. A **FULFIL THE CONTEXT CONDITION** CoCon is an abbreviated `FULFILLING THE CONTEXT CONDITION OF` CoCon.

Not Using 'ONLY'

A context condition indirectly selects an element if the element's context property values match with the context condition. A context property value, e.g. 'Integrate Two Contracts', belongs to a context property name, e.g. 'Workflow'. One context condition can refer to values of several context property names. The CoCon-predicate operation 'ONLY' demands that no other context property values of the same context property name than those specified in the context condition(s) are associated with an constrained element. For example, the inter-value CoCon `ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Integrate Two Contracts' MUST` **ONLY** `BE THE SAME AS ALL COMPONENTS WHERE 'Workflow' CONTAINS 'CustomerMarriage'` demands that a component associated with 'Integrate Two Contracts' must be associated with 'CustomerMarriage'. Due to the CoCon-predicate operation ONLY, it must not have any other values for 'Workflow' than 'Integrate Two Contracts' and 'CustomerMarriage'. I suggest not to use the CoCon-predicate operation ONLY for inter-value CoCons because it easily can be misrepresented. Instead, the type-instance constraint explained in section 4.5.1 can be used to define that certain elements only can be associated with a subset of the valid values of a context property.

Constraints

The following constraints simplify inter-value CoCons:

- The CoCon-predicate operation 'ONLY' is not allowed to be used in inter-value CoCons.
- Both the target set elements and the scope set elements must be of the same metaclass. For instance, it is not allowed to have *components* in the target set and *users* in the scope set of the same inter-value CoCon.
- In case of a `THE SAME AS` CoCon, both the target set elements and the scope set elements must be selected indirectly via one single context condition. Direct selection of constrained elements is not allowed. Thus, the expression *the same as* only refers to the context property values - it means 'in the same context as'. Section 5.6.5 will continue discussing this constraint in detail.
- In case of a `FULFILLING THE CONTEXT CONDITION OF` CoCon, the scope set elements must be selected indirectly via one single context condition.
- It is not allowed to indirectly select scope set or target set elements via the *unrestricted total selection* `ALL ELEMENTS` (see section 3.3.2). Instead, the valid values definition should be used to describe, which values are allowed to be associated with elements.

5.6.3 Detectable Inter-CoCon Conflicts of Inter-Value CoCons

As explained in section 4.3.1, one CoCon can contradict other CoCons. Conflicting requirements can automatically be detected if one of the following rules is violated:

Two Conditions for the Same Element

Let e_{Chico}, e_{Harpo} and $e_{Groucho}$ be elements of the target set or scope set of one inter-value CoCon. Please consider that e_{Chico}, e_{Harpo} and

$e_{Groucho}$ can represent the same (model) element in different target or scope sets. Furthermore, *cp* is a context property name. Among *cp*'s valid values are v_1 and v_2 with $v_1 \neq v_2$. A context condition selects an element because the context property values of this element match the context condition. Hence, the element is selected *due to* a value that is associated with this element, e.g. v_1 or v_2.

1. No element e_{Chico} can be `THE SAME AS` e_{Harpo} due to v_1 if e_{Chico} is `NOT FULFILLING THE CONTEXT CONDITION OF` e_{Harpo} due to v_1.
2. No element e_{Chico} can be `FULFILLING THE CONTEXT CONDITION OF` e_{Harpo} due to v_1 if e_{Chico} is `NOT THE SAME AS` e_{Harpo} due to v_1.
3. No element e_{Chico} can be `NOT FULFILLING THE CONTEXT CONDITION OF` e_{Harpo} due to v_1 if e_{Chico} is `THE SAME AS` e_{Harpo} due to v_1.
4. No element e_{Chico} can be `THE SAME AS` e_{Harpo} ($CoCon_1$) due to v_1 via the context condition cc_1 if $CoCon_2$ refers to v_2 via the context condition cc_2 and defines that e_{Harpo} must be `THE SAME AS` $e_{Groucho}$ and v_1 does not fit cc_2 or v_2 does not fit cc_1

5.6.4 Examples for Using Inter-Value CoCons

Inter-Value Constraints

If the system manages personal data then the valid values of the context property 'Personal Data' explained in section 3.2.3 may be of $VV^{PersonalData}$ = { 'True', 'False'}. This valid values definition demands that no element of the system has other values of 'Personal Data' than 'True' or 'False'. For example, the value 'Possibly' is not allowed. In order to allow it, the valid values definition for 'Personal Data' must be changed. Nevertheless, taking only $VV^{PersonalData}$ in regards can lead to inconsistent context property values because it allows to associate *both* valid values to the same element: according to the valid values, 'Personal Data (Contract Management): { True, False}' is a correct expression. However, this doesn't make sense because the same element cannot have both 'True' and 'False' values. This example illustrates the need for the '*inter-value constraints*' introduced in section 3.2.4: an inter-value constraint must prevent contradicting values of one context property for the same element. Inter-Value CoCons can be used to specify inter-value constraints. In case of 'Personal Data', the following inter-value constraint prevents contradicting values of 'Personal Data' for the same element: `ALL ELEMENTS WHERE 'Personal Data' CONTAINS 'True' MUST NOT BE THE SAME AS ALL ELEMENTS WHERE 'Personal Data' CONTAINS 'False'`.

In this example, the *unrestricted selection* `ALL ELEMENTS` is used to refer to any element regardless of its metaclass.

5.6.5 Related Research on Inter-Value CoCons

(No) Correspondance

This section compares the `THE SAME AS` CoCon-predicate with Model Correspondence Assertions (MoCas) defined [Bus02]: an element e_1 'corresponds to' another element e_2 if both e_1 and e_2 represent the same real world entity. In contrast, `THE SAME AS` predicate only refers to *one* artefact element e that must (not) fulfil both the target set context condition and the scope set context condition. It does not compare e to any other artefact element - not in the same artefact, and not in any other artefact. In `THE SAME AS` CoCons, the constrained elements are only selected via context conditions. Thus, '`THE SAME AS`' means 'in the same context as': it expresses that *one* artefact element does (not) reside in both contexts. In case of MoCas, equality between to elements means that the same real world entity is represented by two *different* artefact elements instead.

OWL Inter-Value CoCons represent knowledge on metadata. Knowledge representation is the field of Artificial Intelligence (AI). Typically, knowledge is represented by characterizing classes of objects and the relationship among them. CoCons express relationships and, thus, are now compared to the ontology language OWL (see [BvHH$^+$04]) developed by the semantic web initiative.

In computer science, an ontology is the attempt to formulate an conceptual schema within a given domain, a typically hierarchical data structure containing all the relevant entities and their relationships and rules (theorems, regulations) within that domain. OWL is an acronym for Web Ontology Language (or Ordinary Wizard Level in Harry Potter). It was developed as a follow-on from RDF (Resource Description Framework) and RDFS, as well as earlier ontology language projects including OIL and DAML. OWL is based on description logic.

In computer science, an **extension** is the set of things to which a property applies, while the **intension** describes the shared properties of a set of extensions. In OWL, the extensions are called **individuals**, while their intensions are called classes. CoCons refer to extensions and call them **elements**. A CoCon expresses a condition on how its constrained elements must relate to each other. This condition is called **CoCon-predicate** and is a relation. In OWL, a relation between two individuals is called a **object property**. CoCons can be expressed in OWL. So, why should we use CoCons if we can use OWL? Besides OWL, many other languages for expressing logic exist which can be used to express CoCons. If you dislike the context-based constraint language CCL defined in [Büb02a] then use your favourite logic language.

In order to express CoCons in a logic language, you have to

- select the constrained elements according to their context and to express relations between them. Typically, logic languages are not used in that way. Instead, they directly specify their constrained elements by naming them.
- define the artefact-type-specific semantics of your logic language for each system artefact you want to monitor for compliance with your expressions. Typically, logic languages are not uses in that way. Instead, they express all knowledge of a system which necessary to check if certain conditions are fulfilled — they specify the same details that are again specified in the relevant system artefacts. On the contrary, CoCons don't need a complete model of the system. They directly check the system artefacts and don't need a logical model that mirrors the information expressed in the system artefacts.

6. UML-Specific Semantics of CCL

This chapter demonstrates how to check a UML model for compliance with the CoCon-predicates proposed in the previous chapter 5. First, section 6.1 explains how to integrate the CoCons of chapter 5 into the UML metamodel because the notion of CoCons is not part of the UML yet. Then, section 6.2 compares CoCons with the UML's standard constraint language OCL. This comparison reveals why CoCons are a new concept.

6.1 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.

6.1.1 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 [OMG03a]), or with the 'constraint' and 'profiles' packages of UML 2.0 (see [OMG03b]).

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 6.1) 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.

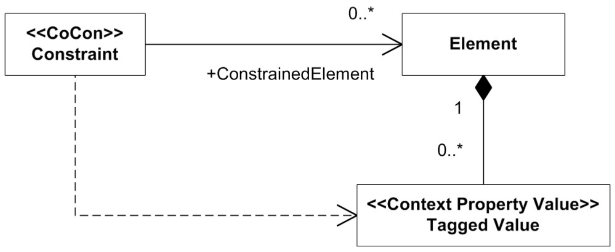

Figure 6.1: A Point Cut indirectly associates a Constraint with its Constrained Elements

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 6.1), the body attribute stores strings that comply with the Context-Based Constraint Language (CCL) syntax definitions given in [Büb02a]. 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.

6.1.2 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* (see section 3.3.8) 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 [OMG03b], 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 element fulfilling its context conditions even if these elements are not associated with each other at all.

In general, a UML constraint hardy 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. UML cannot express such point cut conditions 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 point cut condition anymore, while some new join points may be added to the constrainedElements set. A UML tool which expresses cross-cutting concerns as constraints must re-evaluate the constraint's point cut condition after each model modification in order to keep the list of constrainedElements up to date.

In figure 6.1, 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 point cut condition must be re-evaluated each time when traversing them.

6.2 Comparing Context-Based Constraints with OCL

Typically, the *Object Constraint Language OCL* ([OMG03b, WK99]) is used for the constraint specification of UML models. This section compares OCL to CoCons.

6.2.1 UML Semantics of ACCESSIBLE TO CoCons

This section discusses the translation of `ACCESSIBLE TO` CoCons into OCL via the privacy policy example described in section 2.2 and expressed in CCL in section 3.3.6. 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 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))
     newline .taggedvalue -> 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 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 6.2. 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.

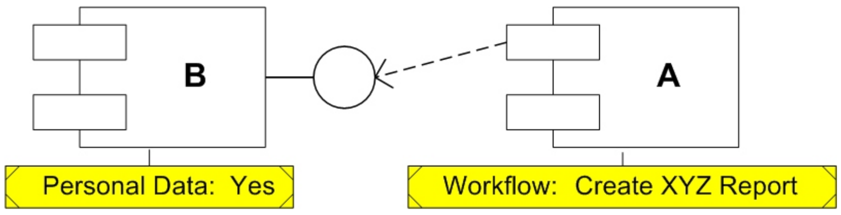

Figure 6.2: The Component 'A' is Not Allowed to Access the Component 'B'

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 OCL constraint specified above is incomplete. 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. Moreover, it does not consider communication between components in a sequence diagrams or collaboration 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. Appendix B on page 121 lists all OCL expressions needed to express the artefact-specific semantics of `ACCESSIBLE TO` CoCons for standard UML 2.0 models.

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): if the component 'B' in figure 6.2 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 [CKM$^+$99a]): '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. This is an example for an artefact-type-specific semantics that increases the complexity of the detect-CoCon-violations algorithm as discussed in section 4.2.2. Recursion can also be handled by deriving context property values from a model element to other model elements as discussed in section 4.5.3.

Different **interpretations** of a CoCon-predicate result in different semantics for the same artefact type. For example, does the CoCon-predicate 'x is accessible to y' refer to composition of components? The answer depends on the interpretation of 'x is accessible to y'. I propose that the artefact-specific semantics should consider composition if the artefact type can represent composition. Moreover, does the CoCon-predicate 'x is accessible to y' refer to recursive invocation between components? I suggest not to consider recursion in the CoCon-predicate. Instead, I propose to define a belongs-to relation between elements that *invoke* each other during the call execution as explained in section 4.5.3. Such belongs-to relations must be considered when evaluating a context condition, but not when evaluating the CoCon-predicate. Nevertheless, recursion can be considered in the CoCon-predicate as demonstrated in appendix B. Again, is depends on the interpretation of 'x is accessible to y'. 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 [CD00] 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). Hence, it must be adapted as demonstrated in appendix B. Without such an adaptation, the OCL expressions do not apply to classes. In that case, the artefact-specific semantics of the `ACCESSIBLE TO`CoCon for UML models simply cannot be applied to those UML models that stick to the UML components approach. Otherwise, applying the (not class-aware) interpretation of 'x is accessible to y' leads to wrong results when running the detect-CoCon-violations algorithm of section 4.2.2. The detect-CoCon-violations algorithm only works if the interpretation of the artefact-specific semantics is correct. This is a limitation of CoCons. On the contrary, the detect-inter-CoCon-conflicts algorithm does not depend on the interpretation of the CoCon-predicate as discussed in section 4.3.2.

In case of `ACCESSIBLE TO` CoCons, the artefact-type-*independent* semantics definition consists of three words: 'is accessible to'. On the contrary, the corresponding artefact-type-specific OCL listing starting on page 124 is many pages long because it considers a lot more details. CCL stays 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 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. Besides the detail level, two other differences between CoCons and OCL are discussed in the next two sections.

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

The Object Management Group (OMG) describes four meta-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' ([CKM+99b]) 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 [RG00], or the OCL constraint must be translated into M_0 level source code as suggested in [HDF00]. 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 version of the privacy policy described in the previous section 6.2.1 must be specified on the *metamodel* level.

On the contrary, M_1 level model elements can 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 tagged values are defined. The tagged values of a model element are available as soon as they are specified on M_1 level. Hence, the model elements constrained by a M_1 level CoCon can already be identified on M_1 level. Nevertheless, identifying the constrained elements is not sufficient. Additionally, each pair of related constrained elements must be checked if it fulfils the CoCon-predicate. Again, the CoCon-predicate can be checked on M_1 level if the semantics definition of the CoCon-predicate refers to M_1 level model elements. 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, a CoCon using these semantics definitions can be expressed on M_1 level and can be checked on M_1 level. This is a major difference between CoCons and the prevailing notion of constraints.

OCL is needed to precisely define the semantics of CCL in UML 2.0 models. However, the person who specifies a CCL expression in a UML *model* must not need to know the artefact-type-specific semantics definitions in OCL at *metamodel* level. Hence, CCL provides a *shortcut*. Without CCL, the designers would have to write down long OCL expressions at metamodel level in order to express the same requirement that can be specified in a short CCL expression on model level. Modifying the *meta*model each time the requirements change is not appropriate.

In [GF94], the following requirements engineering problem is described. 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 [Pal00], 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.

6.2.3 CoCons can Constrain Unassociated Elements

The key concept of CoCons is the indirect selection of constrained 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 models. Every new, changed or removed model 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 with 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.

7. The CCL Analysis & Specification Method CCLM

This chapter introduces general methodical guidance on how to identify and apply CCL. It introduces the **CCL analysis & specification method (CCLM)** that tells how to identify context-based business rules relevant in a certain application domain.

7.1 Overview on CCLM

7.1.1 Background: Adopting Kotonya's and Sommervilles's Requirements Engineering Process

In this section, the different phases and activities of the CCLM are outlined.

Methods

According to [NE00], a method provides a prescription for how to perform a collection of activities, focusing on how a related set of techniques can be integrated, and providing guidance on their use. A method defines the process used to gather requirements, analyse them, and design an application that meets them in every way. There are many methods, each differing in some way or ways from the others. There are many reasons why one method may be better than another for a particular project: For example, some are better suited for large enterprise applications while others are built to design small embedded or safety-critical systems. On another axis, some methods better support large numbers of architects and designers working on the same project, while others work better when used by one person or a small group.

Goal: Identifying CoCons & Context Properties

Gerald Kotonya and Ian Sommerville have defined requirements engineering processes in [KS98]. The CCL Analysis & Specification Method CCLM does not re-invent the wheel. Instead, it refines the process defined in [KS98]. AAccording to requirements engineering process is an organised set of activities that transform inputs to outputs. Inputs to the requirements engineering process are information about existing systems, stakeholder needs, organisational standards, regulations and domain information. Requirements engineering processes vary radically from one organisation to another. Factors contributing to this variability include different technical maturities, different disciplinary involvements, different organisational cultures and different application domains. There is therefore no 'ideal' requirements engineering process. However, most processes include the four requirements engineering phases described in section 1.3: requirements elicitation, requirements negotiation, requirements specification and requirements validation. CCLM adds activities to each of these four phases of the requirements engineering processes as illustrated in figure 7.1. In these additional activities, CCL-specific questions are asked for identifying context properties and context-based constraints. Without adding CCLM's activities to the requirements analysis, the requirements specification documents gathered probably won't assist in identifying context properties and context-based constraints. For each requirements engineering phase, CCL adds some activities:

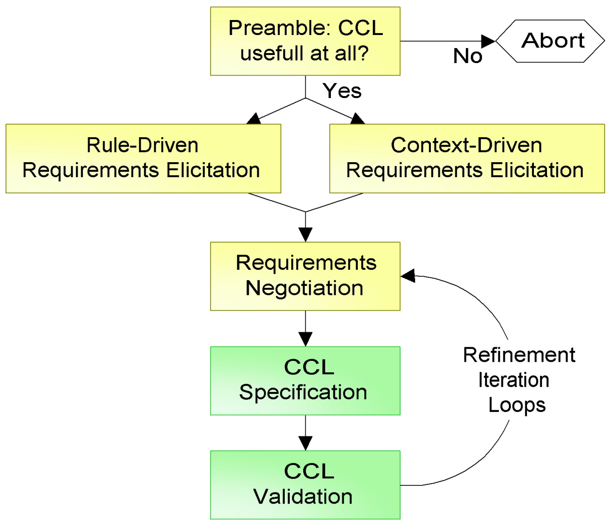
Preamble: CCL usefull at all?
No
Abort
Yes
Rule-Driven Requirements Elicitation
Context-Driven Requirements Elicitation
Requirements Negotiation
CCL Specification
CCL Validation
Refinement Iteration Loops

Figure 7.1: Overview on the Four Phases of CCLM for Identifying Context-Based Business Rules

Four Phases of CCLM

- **Preamble**: The objective of this optional initial phase is to identify quickly whether CCL is useful for a certain software development project at all. If it turns out that CCL is not useful then CCLM ends here. The preamble phase is described in section 7.1.3.
- During **requirements elicitation**, the software requirements are discovered, articulated, and revealed from stakeholders or derived from system requirements. CCLM offers two ways for identifying the appropriate CoCons and context properties: both a rule-driven and a context-driven approach are presented here. As a result of both approaches, informal CoCon-Rules and context properties are identified. The requirements elicitation phase of CCLM is described in section 7.2.
- During **requirements analysis and negotiation**, all stakeholders discuss the informal requirements discovered in the previous phase in order to arrive at a set of agreed upon informal CoCon-Rules and context properties. The requirements negotiation phase of CCLM is explained in section 7.3.
- During **CCL specification**, the informal CoCon-Rules are specified in CCL, and the context properties are applied to the system or its model. The CCL specification phase of CCLM is introduced in section 7.4.
- During **CCL validation**, the CCL expressions are checked for omitted, extra, wrong, ambiguous and inconsistent requirements. Moreover, the compliance of the system or its model with the CCL expressions is validated. The CCL validation phase of CCLM is described in section 7.5.

Iterative Refinement

Managing changing requirements is not only a process of managing documentation, it is also a process of recognising change through continued requirements elicitation, re-evaluation of risk, and evaluation of systems in their operational environment. CCLM does not stick to the top-down waterfall-process model. Instead, it is possible in each CCLM phase to go back to a previous phase in order to refine the results of this previous phase. These forward- and reverse-engineering steps are illustrated in figure 7.2 via tiny arrows between the phases. These arrows from each phase to its previous phase are omitted in figure 7.1 and figure 7.3 in order to keep them comprehensive. Moreover, the first results of the CCLM method don't have to be complete. Instead of completing each development phase in its entirety before advancing to the next, all phases should be revisited multiple times during the project. Hence, the first basic CCL expressions specified during the first CCLM iteration should be iteratively refined later on. In some of the activities described in the oncoming sections, refinement is explicitly addressed. For example, the CCL specification activity offers extra advice for refining each CoCon and prevents misleading CoCons that is only addressed if it is started not for the first time.

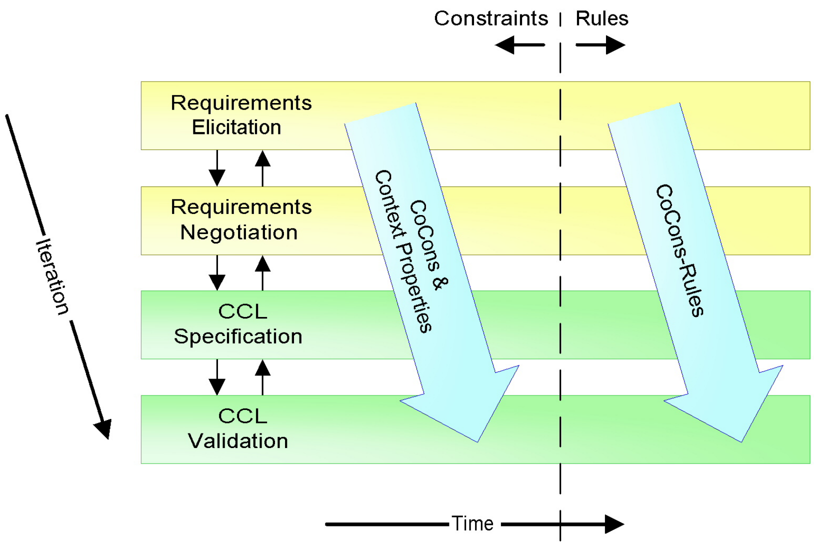

Figure 7.2: Iterative Refinement of CCL Constraints via CCLM

Focus: Constraints

The early iterations of CCLM focus on identifying constraints as illustrated in figure 7.2. The rationale is explained in section 3.4.3: constraints cannot have side effects, while rules can. Hence, the early CCLM iterations should focus on specifying constraints. I suggest ignoring events and actions in the first CCLM iteration. In later iterations, these events and actions can be added to the previously specified CoCons in order to turn them into CoCon-Rules.

7.1.2 Introducing the 11 Activities during the CCLM Process

11 Activities

Most of the phases shown in figure 7.1 consist of several activities. The 11 activities of CCLM are illustrated in figure 7.3. The next sections describe each of these 11 activities in detail. This section provides an overview by outlining each activity:

- The '**Preamble**' activity quickly identifies whether CCL is useful for the particular software development project at all. If it turns out that CCL is not useful then CCLM ends here.
- The CoCons and context properties relevant for a certain application can be identified in two different ways: a rule-driven and a context-driven approach are suggested here. Both approaches

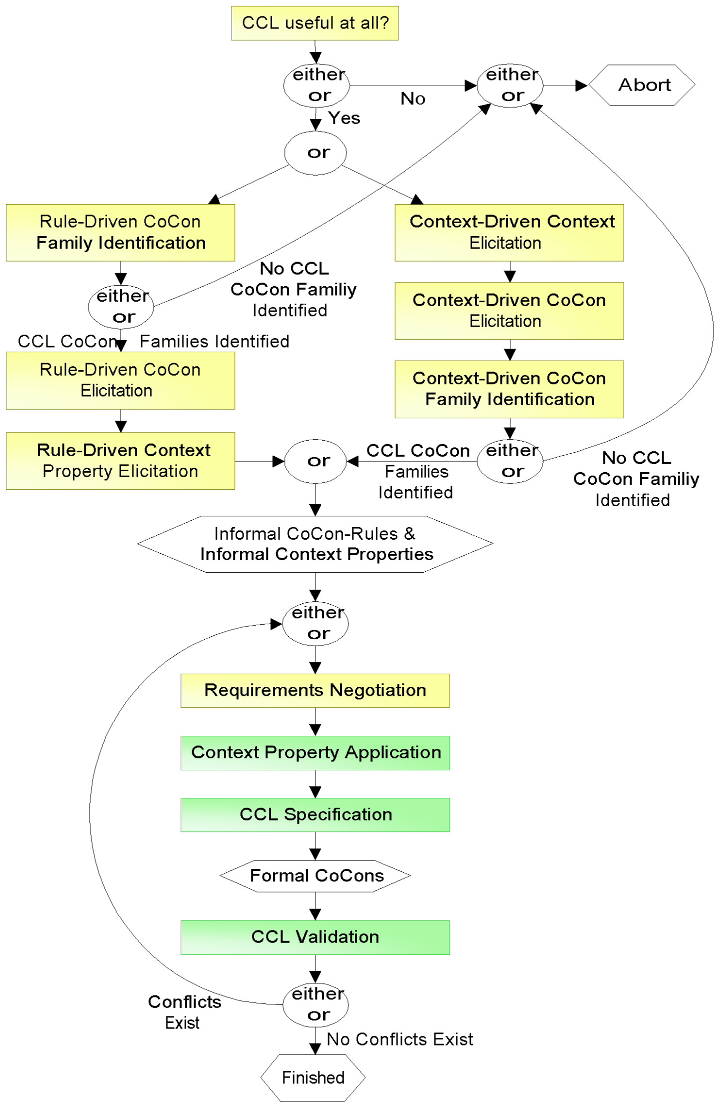
CCL useful at all?
either or
No
either or
Abort
Yes
or
Rule-Driven CoCon Family Identification
Context-Driven Context Elicitation
either or
No CCL CoCon Familiy Identified
Context-Driven CoCon Elicitation
CCL CoCon Families Identified
Rule-Driven CoCon Elicitation
Context-Driven CoCon Family Identification
Rule-Driven Context Property Elicitation
or
CCL CoCon Families Identified
either or
No CCL CoCon Familiy Identified
Informal CoCon-Rules & Informal Context Properties
either or
Requirements Negotiation
Context Property Application
CCL Specification
Formal CoCons
CCL Validation
Conflicts Exist
either or
No Conflicts Exist
Finished

Figure 7.3: CCLM consists of 11 Activities

consist of three activities. It is possible to use either only the rule-driven or only the context driven approach. Best results, however, are achieved by applying both approaches either simultaneously or consecutively as discussed in section 7.2.1.

– The *rule-driven* approach for identifying the relevant CoCons and context properties consists of the following three activities:

  ∗ The '**Rule-Driven CoCon Family Identification**' activity presented in section 7.2.3 helps to identify which five CoCon families are useful at all. Each useless CoCon family must not be considered in the oncoming activities. If it turns out that no CoCon family is of use for the particular application then the rule-driven approach ends here.

  ∗ The '**Rule-Driven CoCon Elicitation**' activity presented in section 7.2.4 assists in discovering the relevant context-based business rules. They are written down as informal CoCon-Rules in plain English without many details.

    * The '**Rule-Driven Context Property Elicitation**' activity introduced in section 7.2.5 facilitates determining the relevant context properties according to the informal CoCon-Rules identified in the previous activity.

  - The *context-driven* approach for identifying the relevant CoCons and context properties consists of the following three activities:

    * The '**Context-Driven Context Property Elicitation**' activity described in section 7.2.7 helps figuring out in which contexts are particularly challenging. In which context do problems arise that need to be handled? For instance, certain situations may call for certain actions.

    * The '**Context-Driven CoCon Elicitation**' activity presented in section 7.2.6 assists in shaping the context-based business rules out of the relevant contexts identified in the previous activity. They are written down as informal CoCon-Rules in plain English that do not reflect many details yet.

    * The '**Context-Driven CoCon Family Identification**' activity presented in section 7.2.3 maps the informal CoCon-Rules identified in the previous activity the CoCon-predicates available in CCL. If none of the informal CoCon-Rules can be mapped into CCL then the context-driven approach ends here.

- During the '**Requirements Negotiation**' activity described in section 7.3, all stakeholders discuss the informal CoCon-Rules identified in the previous activities in order to arrive at a set of agreed upon requirements.

- The **requirements specification** phase of CCLM consists of two activities:

  - The '**Context Property Application**' activity presented in section 7.4.1 applies the context properties identified in the previous activities to the system or its model elements.

  - The informal CoCon-Rules identified and discussed in the previous activities are turned into constraints by removing events and actions and are written down in the formal language CCL during the '**CCL Specification**' activity explained in section 7.4.

- The requirements validation phase of CCLM focuses on the '**Conflict Detection**' activity presented in section 7.5. It checks whether the system (model) enriched with context properties violates the CCL constraints. Moreover, conflicting requirements are detected. If conflicts exist then their solution must be negotiated with the stakeholders as described in section 7.3.

Each activity of CCLM is discussed in one of the following sections.

7.1.3 Preamble: Why to Apply CCL at all?

Before asking any domain specific question, the general question is why to apply CCL to a specific system at all. This preamble lists general reasons for using CCL:

- CCL specifies requirements on an artefact-type-independent level. Therefore, the compliance of a system with requirements expressed via CCL can be validated during modelling, during configuration and at runtime.
- CCL helps to detect conflicting requirements as early as possible. In contrast to OCL constraints, CCL expressions specified during modelling can already be checked on the design level. Fixing conflicting requirements during implementation is much more expensive.
- Requirements or contexts tend to change often. Indirectly selecting the elements involved in a requirement improves adaptability because every new or changed element of the system is constrained automatically if it fits to the context condition of a CCL constraint. Moreover, CCL's capability to indirectly select the elements involved in a requirement is particularly useful in loosely coupled systems or systems that manage frequently changing data. Again, every new or changed element of the system is constrained automatically.
- During design, one requirement can affect several model elements that may not be associated with each other or even belong to different models. At runtime, one requirement can affect several components that may not invoke each other or may even run on different platforms. CCL can express a requirement for a group of otherwise possibly unrelated elements - even across different views, element types, models, or platforms.
- A requirements specification should serve as a document understood by designers, programmers and customers. CCL is an easy comprehensible language that assists English-speaking persons in understanding the system's design rationale. Appendix B demonstrates that CCL is more easily understood than OCL because it ignores unimportant details and because it is similar to plain English.

If none of these issues are important for a certain software system than CCL should not be applied to it. In that case, none of the activities listed in the oncoming sections is relevant.

7.2 The Requirements Elicitation Phase of CCLM

7.2.1 Rules-Driven or Context-Driven Requirements Elicitation

Requirements Elicitation

The first phase of requirements engineering process is elicitation. A general top-down requirements elicitation process consists of the following activities: Identify all stakeholders who could be sources of requirements, e.g. users, customers or domain experts. Then, ask relevant questions in order to gain an understanding of the problem, issues and constraints. Next, analyse the information looking for conflicts, ambiguities, inconsistencies, problems or unresolved issues. Afterwards, confirm your understanding of the requirements with the stakeholders. Finally, create requirements expressions.

The CoCons and context properties relevant in a certain application domain can be identified in two different ways: a rule-driven and a context-driven approach are suggested here.

Goal: Identify Relevant Business Rules

As explained in section 7.1, the objective of CCLM is to find out whether any business rule exists that depends on the context of the system elements or its users. Two strategies for identifying these business rules are introduced here:

Rule-Driven

- The **rule-driven** approach consist of the following three steps:
    1. First, the business experts are asked which of the issues addressed by CCL are needed at all.
    2. Then, the business experts are asked for relevant rules.
    3. Afterwards, these rules are analysed in order to determine the relevant context properties.

Context-Driven

- The **context-driven** approach consist of the following three steps:
    1. First, the business experts are asked in which context that problems arise that need to be handled.
    2. Next, these contexts are analysed in order to determine the relevant CoCon-Rules.
    3. After that, these CoCon-Rules are mapped to CCL.

Difference?

Both strategies aim to identify relevant CoCons and context properties. However, they achieve this goal via different starting points: the rule-driven approach begins asking about rules, and the context-driven approach starts asking about contexts. The rule-driven approach is faster, but reveals less business rules. It is faster because it only addresses business rules that can be expressed in CCL. On the contrary, the context-driven approach identifies useful business rules regardless whether they can be expressed in CCL or not. The context-driven approach may result in business rules that cannot be expressed in CCL. The benefit of the context-driven approach is the *possible identification of future CoCon-predicates.* In that case, the only chance to apply CCL is to add a corresponding CoCon-predicate to CCL as described in section 3.3.5. Alternatively, the requirement may be specified in another formal language or simply written down in plain English.

Pair Requirements Elicitation

It is possible to use either only the rule-driven or only the context-driven approach. Best results, however, are achieved by applying both approaches either simultaneously or consecutively. If more than one interviewer work out the relevant CoCons and context properties of one application, then some interviewer(s) can choose the rule-driven approach for asking the business experts, while the other interviewer(s) can use the context-driven approach in the same time. If only one person is interviewing the business experts, then stills both approaches can be used consecutively. The results of the context-driven interviews can be combined with the results of the rule-driven interviews in order to improve the overall quality of the results as explained in section 7.5.

7.2.2 How to Write Down Informal CoCon-Rules

CCL is best applied to Sets of Elements

Both the rule-driven approach and the context-driven approach suggest to write down **informal CoCon-Rules** in plain English. This section introduced a general guideline how to state these informal CoCon-Rules. CoCons are most useful if they refer to *sets* of system or model elements that are indirectly described via the context in which they are used. Hence, an informal CoCon-Rule should express a business rule for *sets* of elements that are *indirectly* described according to their context. The description of the elements involved in an informal CoCon-Rule should start with 'all' followed by the type of system elements. For instance, the informal CoCon-Rule can refer to 'all components' that are used in a certain context or to 'all users' in a certain context.

Directly Naming Elements

Of course, it is allowed to directly name the elements involved in an informal CoCon-Rule. For instance, the business rule can constrain *the component 'ContractManagement'* or *the user 'Mrs. Molly Million'*. It is also allowed to combine directly named and indirectly described elements. For example, the business rule can constrain *the component 'Contract-Management' and all components needed in the workflow 'Integrate Two Contracts'*.

Hint: From Direct to Indirect Descriptions

In order to identify set elements involved in a business rule it might be easier to write down a list of directly named elements first. If more then one directly named element is addressed in the business rule, then perhaps they share a context. If *all* system elements in this shared context are constrained by this business rule, then the informal CoCon-Rule should describe them indirectly according to their shared context instead of naming all of them directly.

Relate Two Sets of Elements

A CoCon always relates two sets of elements. It relates each element of one set to each element of the other set and defines a condition between each pair of related elements as illustrated in figure 3.3 on page 31. Therefore, the informal CoCon-Rule should describe how all elements of one set relate to all elements of the other set.

Informal Syntax

The informal CoCon-Rule should stick to the following syntax: 'all (description of a set of elements) must (not) be *(fulfilling a certain condition in regards to)* to all (description of a set of elements)'. For instance, an informal CoCon-Rule might state that 'all components handling personal data must *not be accessible* to all users belonging to the controlling department'. In this example, the part in *italics* describes how the elements of one set (certain components) relate to the elements of the other set (certain users) – it defines the CoCon-predicate.

Refinement in Later Iterations

The following questions can be skipped if this is the first CCLM iteration. They refine the results gathered in previous iterations of this activity. An informal CoCon-Rule is easier to handle later on if it expresses the following details:

- When should the system be checked for compliance with the informal CoCon-Rule? Should it be checked during design, during configuration or at runtime? Answering this question can be facilitated by estimating how often the elements involved in the informal CoCon-Rule or their contexts change. Not at all during the system's lifetime, maybe some times a year, or possibly daily?
  - If the involved elements or their contexts change at runtime then the informal CoCon-Rule should be checked at runtime
  - If the involved elements or their contexts change due to (re-)configuration then they should be checked during deployment.
  - If the involved elements or their contexts change due to (re-)-design then they should be checked during modelling.
- For each informal CoCon-Rule write down the rationale why to enforce this CoCon. For instance, write down that 'this requirement

is explained in the companies business plan available in the file thinkahead.doc'.

- Write down what should happen if this business rule is violated. Who is responsible for this action: the software or certain users? For each action, try to figure out if the software can do it, or if human users are needed to do it.

No Syntax

The syntax of an informal CoCon-Rules doesn't matter yet as long as it is comprehensible for English speaking persons. Each informal CoCon-Rule will be mapped into a formal CoCon later on. The business rule should be stated in only one sentence, but the details can be added via extra sentences.

In the next three sections, the rule-driven approach for identifying context-based business rules is presented.

7.2.3 Rule-Driven CoCon Family Identification

This section describes the first activity of the rule-driven approach. It is called the 'Rule-Driven CoCon Family Identification' activity and helps to identify which CoCon families are useful at all. In this activity, the stakeholders are asked the questions in order to find out which CoCon family must (not) be considered in the oncoming activities:

1. Shall all users or all components be able to access all components or business objects? If not, then access permission CoCons should be considered in section 7.2.4.
2. Should certain workflows be protected by transactions? Shall certain parts of the system use time-critical synchronous communication, while others use asynchronous communication, e.g. in message broker applications or in mobile devices? Shall certain communication calls between components be encrypted, logged according to the current context? If any answer is yes here, then communication CoCons should be considered in section 7.2.4.
3. Shall the system be distributed to more than one computer? If yes, then distribution CoCons should be considered in section 7.2.4.
4. Shall the system provide the right information only when it is required and where it is required? If yes then information-need CoCons should be considered in section 7.2.4.
5. Does the system handle a huge number or frequently changing documents, products or data? If yes then inter-value CoCons should be considered in section 7.2.4. Moreover, inter-value CoCons should *always* be considered if any CoCons are used at all.

If the answers to all the questions listed above reveal that no CoCon family is of use at all, then the rule-driven approach ends here. Otherwise, the next activity discovers more details on each useful CoCon family.

7.2.4 Rule-Driven Elicitation of Relevant Informal CoCon-Rules

Identifying CoCons

The 'Rule-Driven CoCon Elicitation' activity presented here assists in discovering the relevant context-based business rules. They are written down as informal CoCon-Rules in plain English that do not reflect many details yet.

Analysing Only Certain CoCon Families

The 22 different CoCon-predicates of CCL are grouped into five CoCon families. The previous activity presented in section 7.2.3 assists in finding out which of the five CoCon families are useful in the application domain at all. This section lists a block of questions for each interesting CoCon family. If a certain CoCon family is uninteresting for the particular application then skip its question block. If the CoCon family is relevant then write down the corresponding business rule down via an informal CoCon-Rule as described in section 7.2.2.

Identifying Relevant Access Permission CoCons

The following questions should not be asked if the application of access permission CoCons was regarded useless in the 'CoCon Family Identification' activity:

- Which users or components are not allowed to access what? Write down this business rule via an informal CoCon-Rule as described in section 7.2.2.
- For each informal CoCon-Rule answer the following question: Does your requirement address general access, or does it particularly address only read access, write access or delete access? If appropriate, refine your initial access permission CoCon. For instance, write down that 'all users belonging to a certain department must not have *read* access to all components that handle certain data'.

Repeat these questions until the business experts cannot identify more access permission requirements.

Identifying Relevant Communication CoCons

The following questions should not be asked if the application of communication CoCons was regarded useless in the 'CoCon Family Identification' activity.

- Which components should be protected via transaction? For instance, it can be useful to protect all components that are needed in a certain workflow in order to protect this workflow.
- If the whole system shall use the same communication technique throughout its lifetime then skip these questions:
  - Which components should communicate synchronously with which other components?
  - Which components should communicate asynchronously with which other components? For instance, message broker applications or mobile devices typically communicate asynchronously.
- Which components should encrypt their communication when calling which other components?
- Which communication calls shall be *logged* according to which context of the caller or callee?
- If an error occurs during a communication call, which calls shall be treated specially?

Repeat these questions until the business experts cannot identify more communication requirements.

Identifying Relevant Distribution CoCons

The following questions should not be asked if the application of distribution CoCons was regarded useless in the 'CoCon Family Identification' activity:

- Which components or business object must (not) be available on which computers? For instance, a workflow can be executed on a certain computer even if the network crashed if all components needed in this workflow are available on this computer. Write down this business rule via an informal CoCon-Rule as described in section 7.2.2.
- For each initial distribution CoCon, try to figure out if the involved components or business objects shall be available as original or as copy. If they are copied, should modifications of the original be propagated to the copies at once or every now and than? Refine you initial distribution CoCon if these questions apply and they can already be answered.

Repeat these questions until the business experts cannot identify more distribution requirements.

Identifying Relevant Information-Need CoCons

The following questions should not be asked if the application of information-need CoCons was regarded useless in the 'CoCon Family Identification' activity:

- Which users should (not) be notified of which documents or business objects? Write down this business rule via an informal CoCon-Rule as described in section 7.2.2.
- If a user is notified of certain documents or business objects, which other documents or business objects are (not) as interesting for every user of the system? If the user should (not) be notified of these other documents or business objects, too, then write it down as informal CoCon-Rule that states which elements are (not) as interesting as which other elements.

Repeat these questions until the business experts cannot identify more information-need requirements.

Identifying Relevant Inter-Value CoCons

The following questions should not be asked if the application of inter-value CoCons was regarded useless in the 'CoCon Family Identification' activity:

- If an element has a certain context property value, which other context property values must it (not) have? Write down this business rule via an informal CoCon-Rule as described in section 7.2.2.
- For each initial inter-value CoCon please state if it is bi-directional (symmetric) or not:
  - A uni-directional inter-value CoCon defines a directed dependency between context property values. For instance, an uni-directional CoCon can state that all books where the 'author'= 'J.K. Rowling' must have the 'genre' = 'fantasy', but not all

books of the of the ‘genre’ = ‘fantasy’ are written by the ‘author’= ‘J.K. Rowling’.

- A bi-directional inter-value CoCon defines a dependency between context property values in both direction. For instance, a bi-directional CoCon can state that all elements where ‘Personal Data’ has the value ‘True’ must not have the value ‘False’ for ‘Personal Data’ and *vice versa*: all elements where ‘Personal Data’ has the value ‘False’ must not have the value ‘True’ for ‘Personal Data’

Repeat these questions until the business experts cannot identify more inter-value requirements.

7.2.5 Rules-Driven Elicitation of Relevant Context Property Candidates

The ‘Rule-Driven Context Property Elicitation’ activity introduced here facilitates determining the relevant context property candidates according to the informal CoCon-Rules identified in the previous activity described in section 7.2.4. A context property candidate may become a context property later on.

For each informal CoCon-Rule identified in the ‘Rule-Driven CoCon Elicitation’ activity the following questions should be asked:

- Does the informal CoCon-Rule describe the elements involved in this business rule indirectly?
- If yes, then are these elements are indirectly selected according to their properties? Are the elements indirectly described via the context in which they are used? In this case, each single criteria for indirectly selecting the elements is a **context property candidate**. Write down the name of this context. It will be more closely discussed later on. Note that one informal CoCon-Rule can use more than one context property candidate. For instance, the requirement ‘all components that handle top-secret data and are used in the field service...’ refers to two context properties. The components are indirectly selected if they handle ‘top-secret data’. Moreover, the components are selected because they belong to the ‘field service’. Both ‘top secret’ and ‘field service’ are context property candidates.

7.2.6 Context-Driven Elicitation of Relevant Context Properties

In the previous sections, the rule-driven approach for identifying the CoCons and context properties relevant in a certain application domain has been described. As discussed in section 7.2.1, another approach exists, too. This and the next sections introduce the context-driven approach for identifying the CoCons and context properties relevant in a certain application domain. The ‘Context-Driven Context Property Elicitation’ activity described here helps figuring out which contexts are particularly challenging. First, context property candidates are identified via the following questions:

- In which context do problems arise that need to be handled?
- Which exceptions or special situations exist that need special treatment? For instance, certain situations may call for certain actions.
- When or in which context do things happen that should be handled?

An element has an endless number of contexts. Only those contexts that need to be handled should be considered. CoCons group elements that share a context. Hence, those context should be considered that facilitate grouping the elements involved in a requirement. A context should be ignored if no requirement exists that refers to this context in order to describe those elements that are involved in the requirement.

Hints: Some Typical Contexts

In order to identify groups of elements to which a business rule applies, the stakeholders should be asked if the typical context property candidates explained in section 3.2.3 apply to their business:

'Workflow' : ...reflects the most frequent workflows in which the associated element is used. Do different workflows exist in the applied business? If no, then the context property 'Workflow' is useless. Otherwise, which are the most important workflows? Which of them are particularly challenging?

'Operational Area' : ...describes, in which department(s) or domain(s) the associated element is used. Do different departments or organisational areas exist in your business? If no, then the context property 'Operational Area' is useless. Otherwise, which are the most important operational areas or departments? In which of them do which problems arise that need to be handled?

'Classified Data' : ... signals whether an element handles confident data. As an example, the context property 'Personal Data' is used throughout this thesis. It signals whether an element handles data of private nature. Do privacy policies or laws exist that apply to certain data managed by your system? If yes, which parts should be handled in which way?

A term that describes the context that should be handled is called context property candidate here. The context property candidates discovered via the questions listed in this section are transformed into context property names and context property values later on as described in section 7.4.1.

7.2.7 Context-Driven Elicitation of Relevant Informal CoCon-Rules

The following questions should be asked for each context property candidate isolated in the previous activity in order to figure out business rules for this context:

- Which condition must be fulfilled between two related elements?
- Which action(s) must be performed when the context property candidate occurs?
- When should the system be checked for compliance with the business rule?

After compiling a list of challenging contexts in the previous activity and figuring out what to do when these contexts occur in this activity, an informal CoCon-Rule should be written down for each action as described in section 7.2.2. This informal CoCon-Rule should refer to the system elements involved via the criteria for deciding whether the context occurs.

After writing down the informal CoCon-Rules, the next section maps them to CCL.

7.2.8 Context-Driven CoCon Family Identification

The activity presented in the previous section results in informal CoCon-Rules. This section describes the third activity of the context-driven approach. It helps to identify which of the informal CoCon-Rules can be expressed in CCL. One of following questions should apply to each informal CoCon-Rule:

1. Does the informal CoCon-Rule demand that certain users or components must (not) be able to access certain other components or business objects? If yes, then it can be specified in CCL via an access permission CoCon. In that case, please refine the informal access permission CoCon-Rule as follows: Does the business rule address general access, or does it particularly address only read access, write access or delete access? If appropriate, refine your initial access permission CoCon-Rule.
2. Does the informal CoCon-Rule demand that certain workflows should be protected by transactions? Does it demand to use time-critical synchronous communication or asynchronous communication, e.g. in message broker applications or in mobile devices? Does it demand to encrypt, log or redirect certain communication calls between components? If any answer is yes here, then it is an informal communication CoCon-Rule. In that case, answer the following questions for each informal communication CoCon-Rule:
   - Which components should be protected via transaction? For instance, it can be useful to protect all components that are needed in a certain workflow in order to protect this workflow.
   - If the whole system shall use the same communication technique throughout its lifetime then skip these questions:
     - Which components should communicate synchronously with which other components?
     - Which components should communicate asynchronously with which other components? For instance, message broker applications or mobile devices typically communicate asynchronously.
   - Which components should encrypt their communication when calling which other components?
   - Which communication calls shall be *logged* according to which context of the caller or callee?
   - If an error occurs during a communication call, which calls shall be treated specially?
3. Does the informal CoCon-Rule demand that to distribute certain components or data to certain computers? If the answer is yes here, then it is an informal distribution CoCon-Rule. In that case, answer the following questions for each informal distribution CoCon-Rule:
   - Which components or business object must (not) be available on which computers? For instance, a workflow can be executed on a certain computer even if the network crashed if all components needed in this workflow are available on this computer.

- Check each initial distribution CoCon, if the involved components or business objects shall be allocated or replicated (copied). If they are replicated, should the be replicated synchronously (time-critical) or asynchronously (better for unstable networks)?

4. Does the informal CoCon-Rule demand that to provide the right information only when it is required and where it is required? If the answer is yes here, then it is an informal information-need CoCon-Rule. In that case, answer the following questions for each informal information-need CoCon-Rule:

   - Which users should (not) be notified of which documents or business objects? Write down this business rule via an informal CoCon-Rule as described in section 7.2.2.
   - If a user is notified of certain documents or business objects, which other documents or business objects are (not) as interesting for every user of the system? Should the user be notified of these other documents or business objects, too?

5. Does the informal CoCon-Rule demand that an element having certain context property values must (not) have certain other context property values? If the answer is yes here, then it is an informal inter-value CoCon-Rule. In that case, answer the following questions for each informal inter-value CoCon-Rule: Moreover, informal inter-value CoCon-Rules should *always* be considered if any CoCons are used at all.

   - If an element has a certain context property value, which other context property values must it (not) have?
   - For each informal inter-value CoCon-Rule please state if it is bi-directional (symmetric) or not:
     - A uni-directional inter-value CoCon defines a directed dependency between context property values. For instance, an uni-directional CoCon can state that all books where the ‘author’= ‘J.K. Rowling’ must have the ‘genre’ = ‘fantasy’, but not all books of the of the ‘genre’ = ‘fantasy’ are written by the ‘author’= ‘J.K. Rowling’.
     - A bi-directional inter-value CoCon defines a dependency between context property values in both direction. For instance, a bi-directional CoCon can state that all elements where ‘Personal Data’ has the value ‘True’ must not have the value ‘False’ for ‘Personal Data’ and *vice versa*: all elements where ‘Personal Data’ has the value ‘False’ must not have the value ‘True’ for ‘Personal Data’

Enhancing CCL? If an informal CoCon-Rule does not fit to any of the questions listed above then CCL is not capable to express this requirements. In that case, maybe a new CoCon-predicate was discovered and can be added to CCL as described in section 3.3.5. If none of the informal business rules fits to the questions listed above then the context-driven approach ends here because CCL cannot express any of the informal business rules.

7.3 The Requirements Negotiation Phase of CCLM

As result of the previous CCLM requirements elicitation phase, informal CoCon-Rules and context property candidates have been written down. The results of the context-driven requirements elicitation can be combined with the results of the rule-driven requirements elicitation in order to improve the overall quality. Before precisely specifying them in a formal language in the next CCLM phase, the informal CoCon-Rules are negotiated between all stakeholders in order to arrive at a set of agreed upon requirements. In order to negotiate informal CoCon-Rules, CCLM adopts a popular model for requirements negotiation: the WinWin negotiation model is based on the Theory W ([FU81, BR89]).

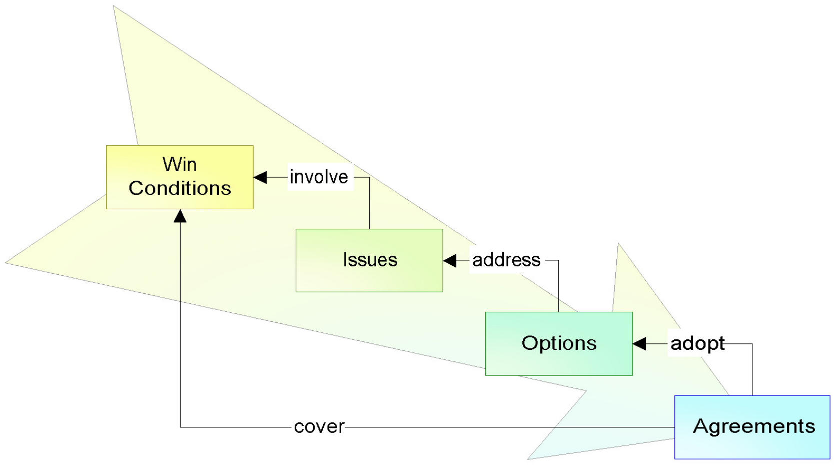

Figure 7.4: The WinWin Negotiation Model

WinWin

WinWin is based on four artefact types as depicted in figure 7.4: Win Conditions, Issues, Options and Agreements. **Win conditions** capture the stakeholder goals and concerns with respect to the new system. An informal CoCon-Rule that has not been negotiated yet is a win condition. If a Win condition is non-controversial, it is adopted by an Agreement. Otherwise, an **Issue** artefact is created to record the resulting conflict among Win Conditions. It should list both the contradicting CCL expressions and the model or system elements involved in the conflict in order to negotiate the contradicting CCL expressions with the stakeholders. One of the benefits of CCL is that violated or contradicting Win condition expressed in CCL can automatically be detected as described in chapter 4. **Options** allow stakeholders to suggest alternative solutions, which address Issues. Finally, **Agreements** may be used to adopt an Option, which resolves the Issue.

Priority

In case of two contradicting CCL expressions, different priorities can be assigned to the CCL expressions involved as follows: a **default CoCon** which applies to all elements where no other CoCon applies should have the lowest priority. Default CoCons can be specified using a total selection as introduced in section 3.3.1. CoCons selecting both their target and their scope set indirectly should have *middle priority*. These constraints express the basic design decisions for *two possibly large* sets of elements. CoCons selecting only element of either the target or the scope set indirectly have *high priority* because they express design decisions for *one possibly large* set of elements. CoCons that select both the target set and the scope set *directly* should have the *highest priority* – they describe exceptions for some individual elements.

Context Conditions

The elements involved in an informal CoCon-Rule can both de directly described via their name or indirectly described via a context condition. An indirect description is called context condition because it refers to the context in which the constrained element is used. For each context condition, the business experts shout be asked the following questions in order to prevent misleading requirements:

- Are the context condition are too general or too precise?
  - A context condition is too general if it describes more elements then it should.
  - A context condition is too precise if it describes less elements then it should.

Choosing Relevant Artefact Types

Different software development processes use different artefact types, e.g. different modelling or programming languages. As described in section 3.3.5, an artefact can be monitored for compliance with CoCons if the artefact-type-specific semantics of those CoCon-predicates used in the requirements specification are defined. The stakeholders should decide which of the artefact types used in the software development process of the planned product shall be monitored for compliance with CoCons. For each artefact type considered as relevant, the artefact-type-specific CoCon semantics must exist. If they do not exist yet than it is expensive to define them. Therefore, it should be negotiated with the stakeholders which artefact type shall be monitored at which expenses. This issue is addressed after requirements elicitation because it depends on the application specific CoCons identified during requirements elicitation. If, for example, all elicitated CoCons only have to be monitored at runtime then only runtime artefacts must be considered when choosing the relevant artefact types.

7.4 The CCL Specification Phase of CCLM

Requirements Specification

Requirements specification is the activity during which the requirements are recorded. In CCLM, requirements specification consists of two activities that record those requirements that can be expressed in CCL: first, context properties are applied to the system (model) as proposed in section 7.4.1. Then, CCL expressions are recorded as suggested in section 7.4.2.

7.4.1 Context Property Application

Analysing Context Property Candidates

In order to identify the relevant context properties, all context property candidates should be written down on one sheet of paper – those that describe a similar context should be grouped by writing them closely together. As explained in section 3.2.2, each context property consists of its name and its allowed values – the context of an element is expressed via metadata formatted as name and value(s).

For each context property candidate identified the business experts are asked questions in order to find out whether it is a context property *name* or a context property *value*:

- If this context property candidate is not similar to any of the other context property candidates, then ask the following questions.

- Is it possible to describe the context property candidate via a more abstract term that is also relevant in the application domain? For instance, 'confidence level' might be a more abstract term for 'top-secret data'. In that case, the more abstract term is the context property name, while the context property candidate is an allowed context property value of this context property name.
- If it is not possible to describe the context property candidate via a more abstract term that is also relevant in the application domain, then the context property candidate is the context property name. Less abstract terms that describe the context more closely will become the values of this context property.

- If this context property candidate is similar to any of the other context property candidates, then ask the following questions.
  - Is it possible to describe the context property candidate via a more abstract term that is also relevant in the application domain? If this more abstract term is among any of the other similar context property candidates then the most abstract term becomes the context property name. If none of the other similar context property candidates is more abstract than the others, then an additional more abstract term must be found. For instance, 'department' might be a more abstract term for 'field service'. In that case, the more abstract term is the context property name, while the context property candidate is an allowed context property value of this context property name.
  - If it is not possible to describe the similar context property candidates via a more abstract term that is also relevant in the application domain, then the context property candidates are not similar and must be handled individually as described above.

Now, the identified context properties should be written on a new sheet of paper in the format 'context property name: allowed context property value(s)'.

Applying the Context Property Values

If no system artefact exists yet, then context property application ends here. Else, the elements of the system artefacts are associated with their context property values. An example is demonstrated in figure 3.1 on page 25. One element can belong to several contexts.

Refinement in Later Iterations

The following part of this section can be skipped if this is the first CCLM iteration. The questions proposed next refine the results gathered in previous iterations of this activity. For each identified context property name the business experts are asked the following questions:

- Who knows details about this context property? In which document(s) are the details defined?
- How can this context be detected? Which criteria exist for deciding whether the context occurs or not?
- Does a software system manage the information for deciding whether an element is in this context or not? If yes, then the current value of this context property for a certain element can be extracted automatically. Automatically extractable values are trustworthier be-

cause they are always up-to-date. The following questions facilitate to find out how the context property value of an element can be extracted:

  - Does the software system manage this property? Do other software systems manage this property? Does the operation system manage this property?
  - Can the current value be calculated from other information managed by software systems? If yes, then write down the algorithm how to calculate the current context property value of an element.

- The following questions address dependencies between context property values:
  - Which values are allowed for this context property? For example, the names of the most frequent workflows in the application domain are allowed values of the context property 'Workflow'.
  - Can all allowed values of this context property be associated with one element without contradicting another context property value? If no, then inter-value constraints between these contradicting context property values exist. Write them down via inter-value CoCons.
  - Does any allowed value of this context property demand that another context property value must be associated with the same element.? If yes, then inter-value constraints between these context property values exist. Write them down via inter-value CoCons.
  - Do certain values of this context property depend on anything? For example, they could depend on the current state of the associated element. If yes, write down these dependency in round brackets behind the dependent context property value as proposed in section 4.5.2.

Define Dependencies

As explained in section 4.5.2, context that depends on information stored in some software system can automatically be extracted from the system each time the context is queried. As a result, the current context value is always available and up-to-date as long as the information on which it depends is available and up-to-date. CCL needs available and up-to-date context property values. Therefore, it is useful to write down how which context of which element can be extracted from where.

Responsibilities

In order to prevent wrong, outdated or missing context property values, a policy could be created that defines who maintains which context properties values. If people know who is responsible for which context property value they hopefully maintain at least those for which they are responsible.

7.4.2 CCL Specification

CoCon Specification

Each informal CoCon-Rule is now expressed in CCL. Each CoCon has a target set and a scope set. These sets contain the constrained elements of the CoCon. We can select the elements of theses sets both directly via their name or indirectly via a context condition. Both direct and indirect selections can be derived from the description of constrained elements in the informal CoCon-Rule. In case of an indirect selection, the context condition should refer to context property values or context property names identified in the previous context property application phase. If the context property used in the informal CoCon-Rule is not listed among the known context properties of this system (model), than step back to the context property application phase and add this context property candidate to the system artefacts.

Refinement in Later Iterations

The following part of this section can be skipped if this is the first CCLM iteration. The questions proposed next refine the results gathered in previous iterations of this activity. Constraints do not tell what happens when a constraint is violated. An expression that also specifies event(s) and action(s) is called rule. A CoCon can be turned into a CoCon-Rule by adding actions and events to it.

Actions

A CoCon relates each element in its target set to each element in its scope set and defines a condition between each pair of related elements. As explained in section 3.3.1, the CoCon attribute `COMPLIANCE-ACTION` describes what action must be taken if two elements are related via this CoCon and comply with the CoCon. It corresponds to the `THEN` part in an `IF-THEN-ELSE` expression. Moreover, the CoCon attribute `VIOLATION-ACTION` specifies what action must be taken if two elements are related via this CoCon but don't comply with the CoCon. It corresponds to the `ELSE` part in an `IF-THEN-ELSE` expression. For each CoCon, the business experts are asked the following questions in order to add actions:

- Who is responsible for deciding upon the action? Is it single or a collaborative decision?
- Which information is required in order to fulfil this action?
- Which `COMPLIANCE-ACTION` shall be executed if two elements are related via this CoCon and fulfil the condition expressed via the CoCon?
- Which `VIOLATION-ACTION` shall be execute if two elements are related via this CoCon but violate the condition expressed via the CoCon?
- Can a software system automatically decide what to do? If yes, write down the criteria.

Events

For each CoCon, the business experts are asked the following questions in order to add events:

- Which events trigger the occurrence of the context? When should the CoCon be checked?
- Is it easier to automatically detect the event than to automatically evaluate the corresponding context condition? In that case, the event should be preferred

7.5 The CCL Validation Phase of CCLM

Requirements validation is the activity during which the requirements are checked for omitted, extra, wrong, ambiguous and inconsistent requirements In the previous requirements specification phase of CCLM, context properties have been associated with the system artefact elements, and CoCons have been specified in CCL. As a result, the algorithms presented in chapter 4 can automatically detect violated or contradicting requirements if the requirements have been expressed in CCL. A system can be validated for whether it complies with the CCL expressions during design, during configuration and at runtime because the CCL expressions are artefact-type-independent. Two prototypical tools for validating UML models during design or Enterprise Java Beans at runtime have been sketched in section 4.4. Automated validation tests provide assurance that the system artefacts meet the requirements specified via CCL.

8. Conclusion

First, section 8.1 will summarize this thesis. Afterwards, section 8.2 will recommend future research topics. Finally, section 8.3 will discuss the limitations, and section 8.4 will explain the benefits of the presented approach.

8.1 Summary

Problems

Checking complex systems for compliance with requirements is difficult because one requirement can affect several possibly unassociated system elements. It is expensive to check which of the system's frequently changing data handled by which of its frequently changing components deployed on which of its frequently changing computers in which of its frequently changing contexts are affected by which requirements.

Goals

In this thesis, I present a new solution for writing down and monitoring crosscutting requirements for complex systems. The objectives of my approach are explained in section 2.3:

- One system consists of different artefact types. My approach does not apply to only one artefact type - it is *independent of artefact types*.
- One system consists of many elements in many artefacts. My approach is *adaptive*: it enables us to identify the elements affected by a requirement automatically.
- My approach enables us to detect violated or contradicting requirements automatically.

Solution

In order to meet these objectives described above, I present the following solution: a context-based constraint (CoCon) can indirectly select its constrained elements according to their context property values. It expresses a condition on how its constrained elements must relate to each other.

In my research I have examined three questions:

- How do we manage context? Section 3.2 has presented a context schema for expressing context. Any context-based approach fails if the context information is wrong, outdated or missing. Therefore , section 4.5 has discussed how to maintain context information.
- How do we write down constraints based on context? Chapter 3 has defined context-based constraints in general, and chapter 5 has proposed 22 different CoCon-predicate for component-based systems. In addition, chapter 7 has provided methodical guidance on how to elicitate and write down contexts and CoCons relevant for a particular application.
- How do we apply context-based constraints after they have been written down? Chapter 4 has examined algorithms for detecting violated or contradicting CoCons in general and has outlined existing

proof-of-concept applications. Moreover, chapter 6 has suggested how to apply CoCons to UML models and has compared CoCons with UML's standard constraint language OCL. This comparison reveals why CoCons are a new concept.

Before discussing the limitations and benefits of my approach, I will suggest future research topics.

8.2 Future Research Recommendations

N-Ary Relations

In order to keep my approach simple, I only examined CoCons relate *two* sets of system elements. One CoCon can express a relation between three or more sets of elements, though. For example, a distribution CoCon $C_d(x, y, z)$ could state that 'any x must be replicated from y to z'. Or, a communication CoCon $C_c(x, y, z)$ could express that 'any x must encrypt its communication with any y about topic z'. Or, an information need CoCon $C_i(x, y, z)$ could demand that 'any user x must be notified of any user y who is notified of z'. Future research that examines CoCons on more than two sets could discuss which n-ary relationships can or should be transformed into binary CoCons. Moreover, it should discuss how to handle the increased complexity of the algorithms for detecting violated or contradicting CoCons.

Grouping CoCons

The grammar of CCL does not consider composition of CoCons for groups of policies. An interesting approach for grouping policies is presented in [Dam02]. CoCons could be adapted to this or other policy grouping approaches.

Events and Actions

Events and actions for CoCon-rules are not deeply considered here. Future research could examine which events and which actions of which CoCon-type are useful for which artefact type.

CoCons in Experts Systems

In chapter 4, algorithms for detecting CoCon-violations or inter-CoCon conflicts have been discussed. They only consider inter-CoCon conflicts between CoCons of one family. Future research could additionally examine inter-CoCon-family conflicts. For instance, can access permission CoCons contradict communication CoCons? Furthermore, CoCons could be applied to expert systems: an inference engine could derive (or infer) additional insights from a knowledge base. Two methods of inference are often used: forward and backward chaining. Forward chaining is a top-down method that takes facts as they become available and attempts to draw conclusions (from satisfied conditions in rules) that lead to actions being executed. Backward chaining is the reverse. It is a bottom-up procedure that starts with goals (or actions) and queries the user about information that may satisfy the conditions contained in the rules. Future research could examine how to use CoCons in forward or backward chaining. For instance, an inference engine could verify inter-CoCon conflicts in order to achieve more insights than those inter-CoCon conflicts discussed in section 4.3.

Fuzzy Contexts

For some contexts it cannot be clearly decided if an element fully resides in this context or not. For example, the time or the location of a context can be fuzzy: when and where did the Russian revolution happen? Different history experts give different and fuzzy answers - they hardly can tell in which second and in which room the Russian revolutions started or ended. Fuzzy logic can assist in expressing fuzzy context. A model for fuzzy temporal context is proposed in [NM03], and a model for fuzzy geographical context is suggested in [JAF03]. Future research could examine how to address fuzzy context models in CoCons.

Applying CoCons during Deployment

During (re-)configuration, a CoCon-aware tool could automatically monitor the system's configuration files for compliance with CCL specifications. For instance, deployment descriptors of Enterprise Java Bean systems can be checked for compliance with CoCons. Among the CoCon families presented here, distribution CoCons and communication CoCon are best checked during deployment. Unfortunately, no case study exists on checking configuration files for compliance with CCL expressions yet.

Enhancing Container Services

Some of the concepts that can be expressed via CoCons are not supported by modern middleware platforms at runtime. If, e.g., a `PROTECTED BY A TRANSACTION WHEN CALLING` CoCon refers to context property values that change at runtime, then this CoCon demands that the transaction mode of the components involved changes at runtime. But, dynamically changing transaction modes are not supported by container services of middleware platforms yet.

8.3 Limitations of CoCons

Trustworthy Context property values

Taking only the context property values of an element into account bears some risks. How can we assure that the context property values are always up-to-date and available? The following approaches can improve the quality of context property values:

- The context property values of an element can automatically be extracted from its software system. If the context property values of an element are extracted newly each time when checked and if the extraction mechanism works correctly, then the context property values are correct and up-to-date as discussed in section 5.5.5. Moreover, the extraction mechanism ensures that context property values are available at all.
- Contradicting context property values can automatically be prevented via inter-value constraints as explained in section 5.6.2.
- Additional Context property values can automatically be derived from existing context property values via belongs-to relations as explained in section 4.5.3.
- Whoever holds the responsibility for the values must be trustworthy. Confidence can be assisted with security techniques as, e.g., public-key infrastructures.
- A policy could defines who maintains which context properties values as suggested in section 7.4.1.

Terminology

Within one system, only one terminology for context property values should be used. For instance, the workflow 'Create Report' should have exactly the same name and the same meaning in every part of the system, even if different companies manufacture or use its parts. Otherwise, a context condition referring to the workflow 'Create Report' might select the too many or too few elements. If more than one terminology for context property values is used, correspondences between heterogeneous context property values can be expressed via correspondence techniques, e.g. Model Correspondence Assertions ([Bus02]). However, not every vocabulary problem can be solved via engineering techniques. These techniques can reduce the heterogeneity, but cannot overcome it completely.

Hence, the need for a controlled terminology remains the key limitation of CoCons.

8.4 Benefits of CoCons

Context Properties 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.

Automatically Detectable Conflicts Algorithms for automatically detecting both violated and contradicting CoCons have been presented. CoCons support the design of software systems from the start of the development process. In contrast to OCL constraints, CoCons specified during modelling can already be checked during modelling at the same metalevel. Hence, the model can be checked for violated or contradicting CoCons already during modelling. Furthermore, inter-CoCon conflicts can even be detected if the precise semantics of the checked system artefact are unknown.

Artefact-Type Independent 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.

Comprehensibility The specification of a system 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. As demonstrated in appendix B, a CoCon can be translated into an artefact-specific constraint which is much longer and much more complicated than the corresponding CoCon. 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.

Coping with Complex Systems 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 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.

Evolution 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 contexts and requirements. According to [MD00], automated support for software evolution is central to solving some very important technical problems in current day software engineering.

Appendix A. The Textual Syntax of CCL

Refined CoCon Syntax

A common syntax for CoCons is defined in section 3.3.2. This appendix refines the common CoCon syntax in order to reflect the 22 CoCon-predicates introduced in chapter 5. Hence, it defines the syntax of the **C**ontext-Based **C**onstraint **L**anguage CCL. It consists of CoCon-predicates for defining requirements for component-based systems.

Syntax of the **C**ontext-Based **C**onstraint **L**anguage CCL

`CoConPredicate`	::=	‘**ACCESSIBLE TO**’ | ‘**READABLE BY**’ | ‘**WRITEABLE BY**’ | ‘**EXECUTEABLE BY**’ | ‘**REMOVEABLE BY**’ | ‘**CACHED WHEN CALLING**’ | ‘**ENCRYPTED WHEN CALLING**’ | ‘**ERRORHANDLED WHEN CALLING**’ | ‘**LOGGED WHEN CALLING**’ | ‘**PROTECTED BY A TRANSACTION WHEN CALLING**’ | ‘**ASYNCHRONOUSLY CALLING**’ | ‘**SYNCHRONOUSLY CALLING**’ | ‘**ALLOCATED TO**’ | ‘**SYNCHONOUSLY REPLICATED TO**’ | ‘**ASYNCHONOUSLY REPLICATED TO**’ | ‘**NOTIFIED OF**’ | ‘**AS INTERESTING AS**’ | ‘**AVAILABLE TO ANYONE INTERESTED IN**’ | ‘**AS INTERESTED AS**’ | ‘**NOTIFIED OF THE SAME AS**’ | ‘**THE SAME AS**’ | ‘**FULFILLING THE CONTEXT CONDITION OF**’
`ElementType`	::=	‘**ELEMENT**’ | ‘**COMPONENT**’ | ‘**CONTAINER**’ | ‘**COMPUTER**’ | ‘**USER**’
`ElementTypes`	::=	‘**ELEMENTS**’ | ‘**COMPONENTS**’ | ‘**CONTAINERS**’ | ‘**COMPUTERS**’ | ‘**USERS**’

Keep it Simple

As explained in section 5.6.2, one the `FULFILLING THE CONTEXT CONDITION OF` CoCons can be abbreviated in order to keep CCL comprehensible for human readers. Hence, the following rule is added to the BNF syntax definition given in section 3.3.2:

Syntax of `FULFIL THE CONTEXT CONDITION` CoCons

`FTCCCoCon`	::=	`TargetSet` ‘**MUST**’ [‘**NOT**’] ‘**FULFIL THE CONTEXT CONDITION**’ `ContextCondition` [‘**WITH**’ `(Attribute)`$*^{AND}$]

Other CoCon-predicates

The list of `CoConPredicate`s and `ElementType`(s) listed above is incomplete. It is tailored for one application domain: only requirements for component-based systems are addressed. Moreover, the `ElementType`(s) only consider a few of the metatypes. For instance, they do no distinguish between `COMPONENT TYPES` and `COMPONENT INSTANCES`. If other `ElementType`(s) or `CoConPredicate`s are important for expressing requirements then the syntax of CCL can be adapted to new application domains by adding the `ElementType`(s) or `CoConPredicate`s to the BNF rules listed above and defining their semantics. Applying CCL to other application domains is not covered here, but it will probably result in additional CoCon-predicates.

Appendix B. Translating Accessability CoCons into OCL

This appendix demonstrates how to map CCL to OCL as sketched in section 6.2.

B.1 Overview

The privacy policy of section 2.2 can be specified in CCL as follows:

ALL COMPONENTS WHERE 'Personal Data' = 'True' `MUST NOT BE ACCESSIBLE TO` *ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Create Report'* .

Example Diagram

As illustrated via the dependency relationship (the dotted arrow) in the UML component diagram shown in figure B.1, component type 'A' invokes component type 'B'. But, the privacy policy CoCon of section 5.2.4 does not allow that the component 'A' invokes the component 'B': the diagram violates the privacy policy.

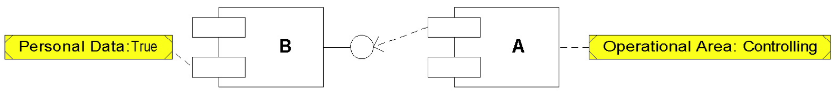

Figure B.1: A Component Diagram Showing Component Types that violate the 'Privacy Policy' CoCon

Metamodel for Component Types in UML 2.0

Figure B.2 shows the metaclasses of the model elements used in figure B.1. Before translating the privacy policy CoCon in an OCL constraint, those metaclasses relevant for the OCL constraint depicted in figure B.2 are quickly explained.

Classifier

A classifier is an element that describes behavioural and structural features. In the metamodel, a Classifier declares a collection of Features, such as Attributes, Methods, and Operations. Classifier is an abstract metaclass.

Component (Type)

A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. It is sometimes called 'component type' here in order to distinguish it from the 'component instance'. A component is shown as a rectangle with two small rectangles protruding from its side.

A UML component diagram shows the dependencies among software components. A component diagram has only a type form, not an instance form. Component instances are defined in UML deployment diagrams. A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships. Components typically expose a set of interfaces, which represent the services provided by the elements that reside on the component. The diagram

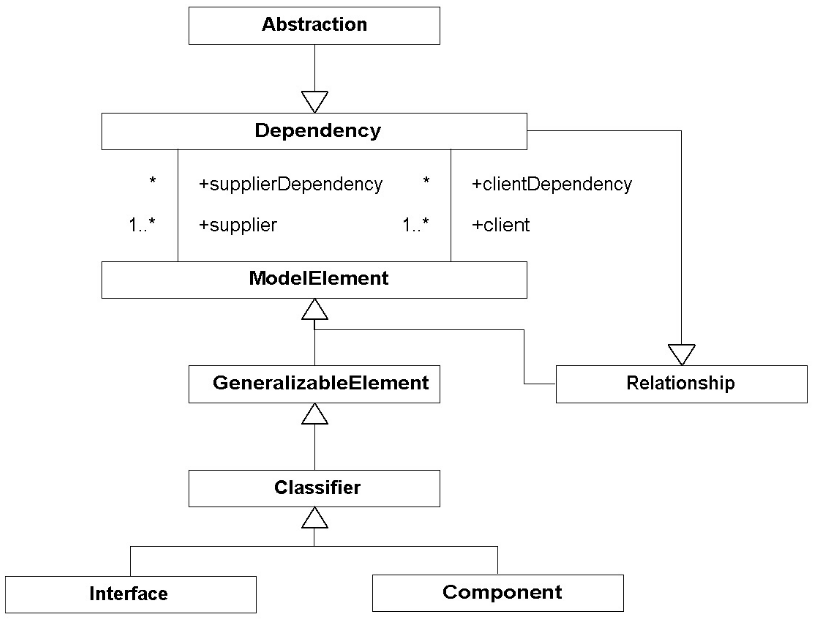

Figure B.2: UML 2.0 Metaclasses For Component Types in Component Diagrams

may show these interfaces and calling dependencies among components, using dashed arrows from components to interfaces on other components.

Interface An interface is a named set of operations that characterize the behaviour of an element. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations. An interface may also be displayed as a small circle with the name of the interface placed below the symbol. The circle may be attached by a solid line (representing an ≪realize≫ abstraction as explained next) to classifiers that support it. A class or component that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. The dashed arrow implies that the class requires no more than the operations specified in the interface; the client class is not required to actually use all of the interface operations. The Realization relationship from a classifier to an interface that it supports is shown by a dashed line with a solid triangular arrowhead (a "dashed generalization symbol"). This is the same notation used to indicate realization of a type by an implementation class.

Dependency A dependency indicates a semantic relationship between two model elements (or two sets of model elements). It relates the model elements themselves and does not require a set of instances for its meaning. It states that the implementation or functioning of one or more elements requires the presence of one or more other elements. In the metamodel, a Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier; that is, the client element requires the presence and knowledge of the supplier element. A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier).

Abstraction

An abstraction is a Dependency relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints.

Realization

One of the UML standard stereotyped classes of Abstraction is Realization (This is the names for the Abstraction class with the stereotypes ≪realize≫ respectively). In figure B.1, the interface of component 'B' is *realized* by component 'B'.:

UML Association enables OCL Navigation

In UML, a binary association is drawn as a solid line connecting two classifier symbols. According to [WK99], each association defines navigation: a shift of attention from one ModelElement to the opposite ModelElement. The name of the navigation is the role name at the opposite end of the association. If a role name is missing, the name of the navigation is the name of the ModelElement (starting with a lower case letter) at that end of the association. The dot-notation is used to reference navigations. The OCL expression presented next uses the dot-notation to navigate along associations between metaclasses.

'naive' OCL

The 'Privacy Policy' CoCon in section 5.2.4 can incompletely be expressed for component (types) in OCL on M_2 (metamodel) level of UML 2.0 as:

```
context   component inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          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 -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()
```

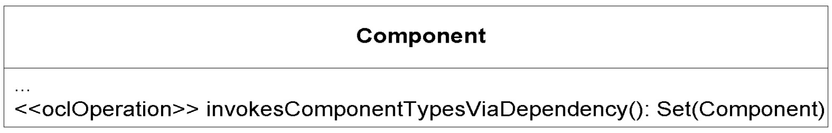
Component
...
<<oclOperation>> invokesComponentTypesViaDependency(): Set(Component)

Figure B.3: Metaclass Component with a Virtual OCL Operation

Using OCL Operations

The expression above consists of three parts. It starts with the translated version of the target set context condition. Then, the artefact-type-specific semantics of `ACESSIBLE TO` CoCons is expressed, before the scope set context condition is specified. The middle part – the `ACCESSIBLE TO` CoCon-predicate semantics– can be expressed as a virtual OCL operation of Component. Expressing the CoCon-predicate via an OCL operation encapsulates the important part of the OCL expression given above in order to reuse this part later on in other OCL expressions. This virtual operation is illustrated in figure B.3. The '...' dots indicate, that the other properties of the metaclass Component are not changed. According to [WK99], the stereotype ≪oclOperation≫ defines two things about an operation:

- The operation is added to the (meta-)model for the purpose of using it in OCL expressions
- The operation does not need to appear in any instance of this meta-class. It is use for OCL specification purpose only

OCL operation for Component Type Dependencies

The following OCL operation specifies the artefact-type-specific semantics of `ACCESSIBLE TO` CoCons for component types:

```
 context  component::invokedComponentTypesViaDependency() :
          Set(Component)
   post:  result = 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))
```

OCL for Component Type Dependencies

This OCL operation can be applied in order to specify a part of the ‘Privacy Policy’ (see section 5.2.4) in OCL:

```
 context  component inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
 implies  self.invokedComponentTypesViaDependency()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()
```

a implies b

The ‘implies’ operation used above states that when the first part is true then the second part must be true. This OCL expression covers only a tiny part of the semantics of the ‘Privacy Policy’ CoCon in section 5.2.4, though. The missing parts are discussed in the next sections.

B.2 Artefact-Specific Semantics for UML

The previous section has explained the concept of OCL operations. This section lists all OCL operations needed to map the ‘Privacy Policy’ CoCon of section 5.2.4 into OCL without explaining each OCL operation.

Component Types via Dependency

```
 context  component::invokedComponentTypesViaDependency() :
          Set(Component)
   post:  result = 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))
```

Component Instances via Dependency

```
context  componentinstance::invokedComponentInstancesViaDependency()
         : Set(ComponentInstance)
  post:  result = self.classifier
         -> select(c | c.oclIsTypeOf(Component))
         .invokedComponentTypesViaDependency()
         .instance -> select(i | i.oclIsKindOf(ComponentInstance))
```

Component Instances via Message

```
context  componentinstance::invokedComponentInstancesViaMessage()
         : Set(ComponentInstance)
  post:  result = self.playedRole
         .message.receiver
         .conforminginstance -> select(c |
         c.oclIsTypeOf(ComponentInstance) )
```

Component Instances via Stimulus

```
context  componentinstance::invokedComponentInstancesViaStimulus()
         : Set(ComponentInstance)
  post:  result = self.stimulus
         .receiver -> select(c | c.oclIsTypeOf(ComponentInstance)
         )
```

Component Instances via Link

```
context  componentinstance::invokedComponentInstancesViaLink() :
         Set(ComponentInstance)
  post:  result = self.linkEnd.link
         .stimulus.communicationLink.connection
         .instance -> select(c | c.oclIsTypeOf(ComponentInstance)
         )
```

UML Components

The next OCL statements refer to classes or objects (instances of classes) because the UML components approach described in [CD00] uses stereotyped classes to model components. Hence, the OCL statemments referring to classes or objects are only needed if the UML Components approach is used.

Classes via Dependend Interface

```
context  class::invokedClassesViaDependendInterface() :
         Set(Class)
   pre:  self.stereotype.baseClass = "comp spec"
  post:  result = 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(class)
            and c.stereotype.baseClass = "comp spec")
```

Classes via Dependend Class

```
context  class::invokedClassesViaDependendClass() :  Set(Class)
   pre:  self.stereotype.baseClass = "comp spec"
  post:  result = self.clientDependency.supplier
         -> select(i | i.oclIsTypeOf(Class)
            and i.stereotype.name = "interface type")
         .clientDependency
         -> select(d | d.oclIsKindOf(Abstraction)
            and d.stereotype.name = "realize"
            and d.supplier.oclIsKindOf(Classifier))
         .supplier -> select(c | c.oclIsTypeOf(class)
            and c.stereotype.baseClass = "comp spec")
```

Classes via Dependency

```
context  class::invokedClassesViaDependency() :  Set(Class)
  post:  result = self.invokedClassesViaDependendClass()
         ->union(self.invokedClassesViaDependendInterface())
```

Objects via Dependency

```
context  object::invokedCompSpecObjectsViaDependency() :
         Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = 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(object)
             and c.classifier.stereotype.baseClass = "comp spec")
```

Objects via Message

```
context  object::invokedObjectViaMessage() :  Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = self.playedRole
         .message.receiver
         .conforminginstance -> select(c | c.oclIsTypeOf(Object)
             and c.classifier.stereotype.baseClass = "comp spec")
```

Object via Stimulus

```
context  object::invokedObjectViaStimulus() :  Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = self.stimulus
         .receiver -> select(c | c.oclIsTypeOf(Object)
             and c.classifier.stereotype.baseClass = "comp spec")
```

Object via Link

```
context  object::invokedObjectInstancesViaLink() :  Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = self.linkEnd.link
         .stimulus.communicationLink.connection
         .instance -> select(c | c.oclIsTypeOf(Object)
             and c.classifier.stereotype.baseClass = "comp spec")
```

Composition

Composed components can be expressed via UML. Therefore, the artefact-specific semantics of `ACCESSIBLE TO` CoCons for UML models need to consider composition, too.

Composed Component Types via Dependency

```
context  component::
            invokedComposedComponentTypesViaDependency() :
         Set(Component)
  post:  result = self.invokedComponentTypesViaDependency()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
            -> select(c | c.oclIsTypeOf(Component)))
```

Composed Classes via Dependency

```
context  class::
            invokedComposedClassesViaDependency() :  Set(Class)
   pre:  self.stereotype.baseClass = "comp spec"
  post:  result = self.invokedClassesViaDependency()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
            -> select(c | c.oclIsTypeOf(class)))
```

Composed Component Instances via Dependency

```
context  componentinstance::
           invokedComposedComponentInstancesViaDependency() :
         Set(ComponentInstance)
  post:  result = self.invokedComponentInstancesViaDependency()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
           -> select(c | c.oclIsTypeOf(componentinstance)))
```

Composed Objects via Dependency

```
context  object::
           invokedComposedObjectsViaDependency() :  Set(Object)
  post:  result = self.invokedObjectsViaDependency()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
           -> select(c | c.oclIsTypeOf(Object)))
```

Composed Objects via Message

```
context  object::
           invokedComposedObjectViaMessage() :  Set(Object)
   pre:  self.invokedObjectViaMessage()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
           -> select(c | c.oclIsTypeOf(object)))
```

Composed Objects via Stimulus

```
context  object::
           invokedComposedObjectViaStimulus() :  Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = self.invokedObjectViaStimulus()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
           -> select(c | c.oclIsTypeOf(object)))
```

Composed Objects via Link

```
context  object::
           invokedComposedObjectInstancesViaLink() :  Set(Object)
   pre:  self.classifier.stereotype.baseClass = "comp spec"
  post:  result = self.invokedObjectInstancesViaLink()
         ->union(associationEnd
             -> select(a | a.aggregation = "composite")
             .participant
           -> select(c | c.oclIsTypeOf(object)))
```

Component Types via Rekursive Dependency

```
context  component::
           invokedComponentTypesViaDependency-up-to(n) :
         Set(Component)
  post:  result =
    if   (n==1)
  then   invokedComponentTypesViaDependency()
  else   invokedComponentTypesViaDependency()
         ->union(invokedComponentTypesViaDependency()
           .invokedComponentTypesViaDependency-up-to(n-1))
         Nat->forall(n |
             invokedComponentTypesViaDependency-up-to(n) =
             invokedComponentTypesViaDependency-up-to(n+1)
         implies invokedComponentTypesViaDependency() =
             invokedComponentTypesViaDependency-up-to(n) )
```

Recursively Composed Component Types

```
context  component::
            invokedComposedComponentTypesViaDependency-up-to(n)
            :  Set(Component)
  post:  result =
     if  (n==1)
   then  invokedComposedComponentTypesViaDependency()
   else  invokedComposedComponentTypesViaDependency()
         ->union(invokedComposedComponentTypesViaDependency()
            .invokedComposedComponentTypesVia-
         Dependency-up-to(n-1))
         Nat->forall(n |
             invokedComposedComponentTypesViaDependency-up-to(n) =
             invokedComposedComponentTypesViaDependency-up-to(n+1)
         implies invokedComposedComponentTypesViaDependency() =
             invokedComposedComponentTypesViaDependency-up-to(n) )
```

Classes via Recursive Dependency

```
context  class::
            invokedClassesViaDependency-up-to(n)
            :  Set(Class)
  post:  result =
     if  (n==1)
   then  invokedClassesViaDependency()
   else  invokedClassesViaDependency()
         ->union(invokedClassesViaDependency()
            .invokedClassesViaDependency-up-to(n-1))
         Nat->forall(n |
             invokedClassesViaDependency-up-to(n) =
             invokedClassesViaDependency-up-to(n+1)
         implies invokedClassesViaDependency() =
             invokedClassesViaDependency-up-to(n) )
```

Recursively Composed Classes via Dependency

```
context  class::
            invokedComposedClassesViaDependency-up-to(n)
            :  Set(Class)
  post:  result =
     if  (n==1)
   then  invokedComposedClassesViaDependency()
   else  invokedComposedClassesViaDependency()
         ->union(invokedComposedClassesViaDependency()
            .invokedComposedClassesViaDependency-up-to(n-1))
         Nat->forall(n |
             invokedComposedClassesViaDependency-up-to(n) =
             invokedComposedClassesViaDependency-up-to(n+1)
         implies invokedComposedClassesViaDependency() =
             invokedComposedClassesViaDependency-up-to(n) )
```

Component Instances via Recursive Dependency

```
context  componentinstance::
            invokedComposedComponentInstancesViaDependency-up-to(n)
            :  Set(Componentinstance)
  post:  result =
     if  (n==1)
   then  invokedComposedComponentInstancesViaDependency()
   else  invokedComposedComponentInstancesViaDependency()
         ->union(invokedComposedComponentInstancesViaDependency()
            .invokedComposedComponentInstancesViaDependency-up-to(n-1))
         Nat->forall(n |
             invokedComposedComponentInstancesViaDependency-up-to(n)
         =
             invokedComposedComponentInstancesViaDependency-up-to(n+1)
         implies invokedComposedComponentInstancesViaDependency()
         =
             invokedComposedComponentInstancesViaDependency-up-to(n)
         )
```

Objects via Recursive Dependency

```
context  object::
            invokedObjectsViaDependency-up-to(n)
            :  Set(Object)
  post:  result =
     if  (n==1)
   then  invokedObjectsViaDependency()
   else  invokedObjectsViaDependency()
         ->union(invokedObjectsViaDependency()
            .invokedObjectsViaDependency-up-to(n-1))
         Nat->forall(n |
             invokedObjectsViaDependency-up-to(n) =
             invokedObjectsViaDependency-up-to(n+1)
         implies invokedObjectsViaDependency() =
             invokedObjectsViaDependency-up-to(n) )
```

Objects via Recursive Message

```
context  object::
            invokedObjectViaMessage-up-to(n)
            :  Set(Object)
  post:  result =
     if  (n==1)
   then  invokedObjectViaMessage()
   else  invokedObjectViaMessage()
         ->union(invokedObjectViaMessage()
            .invokedObjectViaMessage-up-to(n-1))
         Nat->forall(n |
             invokedObjectViaMessage-up-to(n) =
             invokedObjectViaMessage-up-to(n+1)
         implies invokedObjectViaMessage() =
             invokedObjectViaMessage-up-to(n) )
```

Recursively Composed Objects via Dependency

```
context  object::
            invokedComposedObjectViaMessage-up-to(n)
            :  Set(Object)
  post:  result =
    if   (n==1)
  then   invokedComposedObjectViaMessage()
  else   invokedComposedObjectViaMessage()
         ->union(invokedComposedObjectViaMessage()
            .invokedComposedObjectViaMessage-up-to(n-1))
         Nat->forall(n |
             invokedComposedObjectViaMessage-up-to(n) =
             invokedComposedObjectViaMessage-up-to(n+1)
         implies invokedComposedObjectViaMessage() =
             invokedComposedObjectViaMessage-up-to(n) )
```

Objects via Recursive Stimulus

```
context  object::
            invokedObjectViaStimulus-up-to(n)
            :  Set(Object)
  post:  result =
    if   (n==1)
  then   invokedObjectViaStimulus()
  else   invokedObjectViaStimulus()
         ->union(invokedObjectViaStimulus()
            .invokedObjectViaStimulus-up-to(n-1))
         Nat->forall(n |
             invokedObjectViaStimulus-up-to(n) =
             invokedObjectViaStimulus-up-to(n+1)
         implies invokedObjectViaStimulus() =
             invokedObjectViaStimulus-up-to(n) )
```

Recursively Composed Objects via Stimulus

```
context  object::
            invokedComposedObjectViaStimulus-up-to(n)
            :  Set(Object)
  post:  result =
    if   (n==1)
  then   invokedComposedObjectViaStimulus()
  else   invokedComposedObjectViaStimulus()
         ->union(invokedComposedObjectViaStimulus()
            .invokedComposedObjectViaStimulus-up-to(n-1))
         Nat->forall(n |
             invokedComposedObjectViaStimulus-up-to(n) =
             invokedComposedObjectViaStimulus-up-to(n+1)
         implies invokedComposedObjectViaStimulus() =
             invokedComposedObjectViaStimulus-up-to(n) )
```

Objects via Recursive Link

```
context  object::
            invokedObjectInstancesViaLink-up-to(n)
            :  Set(Object)
  post:  result =
     if  (n==1)
   then  invokedObjectInstancesViaLink()
   else  invokedObjectInstancesViaLink()
         ->union(invokedObjectInstancesViaLink()
            .invokedObjectInstancesViaLink-up-to(n-1))
         Nat->forall(n |
             invokedObjectInstancesViaLink-up-to(n) =
             invokedObjectInstancesViaLink-up-to(n+1)
         implies invokedObjectInstancesViaLink() =
             invokedObjectInstancesViaLink-up-to(n) )
```

Recursively Composed Objects via Link

```
context  object::
            invokedComposedObjectInstancesViaLink-up-to(n)
            :  Set(Object)
  post:  result =
     if  (n==1)
   then  invokedComposedObjectInstancesViaLink()
   else  invokedComposedObjectInstancesViaLink()
         ->union(invokedComposedObjectInstancesViaLink()
            .invokedComposedObjectInstancesViaLink-up-to(n-1))
         Nat->forall(n |
             invokedComposedObjectInstancesViaLink-up-to(n) =
             invokedComposedObjectInstancesViaLink-up-to(n+1)
         implies invokedComposedObjectInstancesViaLink() =
             invokedComposedObjectInstancesViaLink-up-to(n) )
```

These OCL expressions do not consider any recursively composed components via any recursive invocation in order not to spoil too many pages.

B.3 The Privacy Policy in OCL

The Privacy Policy CoCon of section 5.2.4 can be mapped into the following OCL expression that use the OCL operations defined in the previous section.

```
context  component inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComponentTypesViaDependency()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()
```

```
context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaDependency()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaMessage()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaStimuli()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaLink()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaMessage()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()
```

```
context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaStimulus()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   componentinstance inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedComponentInstancesViaLink()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   component inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedClassesViaDependency()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   object inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedCompSpecObjectsViaDependency()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()

context   object inv:
          self.taggedvalue->select(tv | tv.dataValue = "Create
          Report")
          .type -> select(td | td.name = "Workflow")
          -> notEmpty()
implies   self.invokedObjectViaMessage()
          .taggedvalue -> select(tv | tv.dataValue = "True")
          .type -> select(td | td.name = "Personal Data")
          -> Empty()
```

```
context  object inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedObjectViaStimulus()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  object inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedObjectInstancesViaLink()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  component inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedComponentTypesViaDependency()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  class inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedClassesViaDependency()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  componentinstance inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedComponentInstancesViaDependency()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()
```

```
context  class inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedObjectsViaDependency()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  object inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedObjectViaMessage()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  object inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedObjectViaStimulus()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()

context  object inv:
         self.taggedvalue->select(tv | tv.dataValue = "Create
         Report")
         .type -> select(td | td.name = "Workflow")
         -> notEmpty()
implies  self.invokedComposedObjectInstancesViaLink()
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()
```

Bibliography

[AB93] Robert Arnold and Shawn Bohner. Impact analysis - towards a framework for comparison. In *International Conference on Software Maintenance (ICSM)*, pages 292–301. IEEE CS Press, september 1993.

[AB98] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis.* IEEE Computer Society Press, 1998.

[Ada79] Douglas Neil Adams. *Hitchhikers Guide to the Galaxy.* Pan Books, London, 1979.

[AMBR02] João Araújo, Ana Moreira, Isabel Brito, and Awais Rashid. Aspect-oriented requirements with UML. In Mohamed Kandé, Omar Aldawud, Grady Booch, and Bill Harrison, editors, *Workshop on Aspect-Oriented Modeling with UML*, 2002.

[Ang92] Peter A. Angeles. *Harper Collins Philosophy Dictionary.* HarperCollins, New York, 1992.

[AS96] Varol Akman and Mehmet Surav. Steps toward formalizing context. *AI Magazine*, 17(3):55–72, 1996.

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

[Bak00] Thomas Baker. A grammar of dublin core. *D-Lib Magazine*, 6(10):47–60, october 2000.

[BB05] Felix Bübl and Michael Balser. Tracing cross-cutting requirements via context-based constraints. In Hongji Yang, editor, 9^{th} *Conference on Software Maintenance and Reengineering, Manchester, Great Britain.* IEEE computer, March 2005.

[BBB$^+$00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, 2000.

[BC04] Elisa Baniassad and Siobhán Clarke. Finding aspects in requirements with theme/doc. In *Early Aspects Workshop*, 2004.

[Bec98] Ulrich Becker. D^2AL - a design-based distribution aspect language. Technical Report TR-I4-98-07 of the Friedrich-Alexander University Erlangen-Nürnberg, 1998.

[Ber02] Caroline Berthomieu. Matrix of propagation - a concept to trace the impact of modifications on software components. Technical Report of the Fraunhofer ISST, Germany, November 2002.

[BFG+93] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hussmann, Dieter Nazareth, Oscar Slotosch, Franz Regensburger, and Ketil Stølen. The requirement and design specification language SPECTRUM, an informal introduction (version 1.0), part 1 & 2. Technical Report TUM-I9312, Technical University Munich, 1993.

[BGJ99] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In B.Rumpe and R.B.France, editors, *2nd International Conference on the Unified Modeling Language, Colorado, USA*, volume 1723 of *LNCS*, pages 249–264. Springer, 1999.

[Bil02] Alexander Bilke. Erweiterbare Proxyarchitekturen für Komponenteninfrastrukturen. Diploma Thesis, Technical University Berlin, Germany, Research Group CIS, May 2002.

[BK05] Daniel M. Berry and Erik Kamsties. The syntactically dangerous all and plural in specifications. *IEEE Software*, 22(1):55–57, January 2005.

[BL01] Felix Bübl and Andreas Leicher. Desiging Distributed Component-Based Systems With DCL. In 7^{th} *IEEE Intern. Conference on Engineering of Complex Computer Systems ICECCS, Skövde, Sweden*, pages 144–154. IEEE Computer Soc. Press, June 2001.

[BL02] Felix Bübl and Andreas Leicher. Überwachung von Anforderungen an Komponenten. *OBJEKTspektrum*, 4:67–72, July/August 2002.

[BR89] Barry W. Boehm and Rony Ross. Theory w software project management: Principles and examples. *IEEE Transactions on Software Engineering*, 15(7):902–916, 1989.

[Bre94] Francis Bretherton. Reference model for metadata: A strawman. DRAFT 3/2/94, University of Wisconsin, 1994.

[BS02] Paolo Bouquet and Luciano Serafini. Comparing formal theories of context in AI. Technical Report IRST 0201-02, Istituto Trentino di Cultura, January 2002.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[Büb00a] Felix Bübl. Context properties for the flexible grouping of model elements. In Hans-Joachim Klein, editor, *12. GI Workshop 'Grundlagen von Datenbanken'*, Technical Report Nr. 2005, pages 16–20. Christian-Albrechts-Universität Kiel, June 2000.

[Büb00b] Felix Bübl. Towards desiging distributed systems with ConDIL. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, volume 1999 of *LNCS*, pages 61–79, Berlin, November 2000. Springer.

[Büb02a] Felix Bübl. The context-based constraint language CCL for components. Technical Report 2002-20, Technical University Berlin, Germany, October 2002.

[Büb02b] Felix Bübl. Introducing context-based constraints. In Herbert Weber and Ralf-Detlef Kutsche, editors, *Fundamental Approaches to Software Engineering (FASE '02), Grenoble, France*, volume 2306 of *LNCS*, pages 249–263, Berlin, April 2002. Springer.

[Büb03] Felix Bübl. What must (not) be available where? In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *5th International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[Büb05] Felix Bübl. Never mind the source code, but be aware of the context when dealing with cross-cutting requirements. In *Early Aspects Workshop*, October 2005.

[Bus02] Susanne Busse. Modellkorrespondenzen für die kontinuierliche Entwicklung mediatorbasierter Informationssysteme. PhD Thesis, Technical University Berlin, Germany, Logos Verlag, 2002.

[BvHH$^+$04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. Technical report, W3C, Feb 2004.

[CD00] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[CGH92] Stefan Conrad, Martin Gogolla, and Rudolf Herzig. Troll light: A core language for specifying objects. Technical Report Technical Report 92-06, Technical University Braunschweig, 1992.

[CHJK02] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan. Specification, implementation, and deployment of components. *Communications of the ACM*, 45(10):35–40, 2002.

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[CKM$^+$99a] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, , and Alan Wills. The amsterdam manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, 1999.

[CKM$^+$99b] Steve Cook, Anneke Kleppe, Richard Mitchell, Jos Warmer, and Alan Wills. Defining the context of OCL expressions. In B.Rumpe and R.B.France, editors, *2nd International Conference on the Unified Modeling Language, Colorado, USA*, volume 1723 of *LNCS*. Springer, 1999.

[Cla02] Siobhán Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1):71–100, 2002.

[CNYM00] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic, Boston, 2000.

[CSN$^+$96] H. C. Chen, B. Schatz, T. Ng, J. Martinez, A. Kirchhoff, and C. T. Lin. A parallel computing approach to creating engineering concept spaces for semantic retrieval - the illinois digital library initiative project. *Ieee Trans. On Pattern Analysis And Machine Intelligence*, 18:771–782, 1996.

[Dam02] Nicodemos Damianou. A policy framework for management of distributed systems. PhD Thesis, Imperial College, London, UK, 2002.

[Dav90] Alan M. Davis. *Software Requirements: Analysis and Specification*. Prentice Hall, Englewood, Cliffs, NK,, 1990.

[Dev91] Keith Devlin. *Logic and Information*. Cambridge University Press, New York, 1991.

[Dey01] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.

[Dij68] Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1990.

[FU81] Roger Fischer and William Ury, editors. *Getting to Yes: Negotiation Agreement Without Giving In*. Penguin Books, New York, 1981.

[GF94] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering (ICRE '94), Colorado Springs, Colorado, U.S.A.*, pages 94–101. IEEE Computer Society Press, April 1994.

[GM78] S.J. Greenspan and C.L. McGowan. Structuring software development for reliability. *Microelectronics and Reliability*, 17:75–84, January 1978.

[Har02] Christoph Hartwich. Flexible distributed process topologies for enterprise applications. In *Proc. of the 3rd Intl. Workshop on Software Engineering and Middleware (SEM 2002*, Orlando, Florida, May 2002.

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran 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.

[Hir00] Graeme 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.

[ISO00] ISO/IEC. Information technology - software product quality -part 1: Quality model. FDIS 9126-1, 2000.

[JAF03] Christopher B. Jones, Alia I. Abdelmoty, and Gaihua Fu. Maintaining ontologies for geographical information retrieval on the web. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *International Conference on Ontologies, Databases, and Application of Semantics (ODBASE), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming ECOOP*, volume 1241 of *LNCS*, pages 220–242, Berlin, 1997. Springer.

[KM97] Jeff Kramer and Jeff Magee. Exposing the skeleton in the coordination closet. In *Coordination 97, Berlin*, pages 18–31, 1997.

[KMM+97] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. GroupLens: applying collaborative filtering to usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[KR04] Shmuel Katz and Awais 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.

[KS96] Vipul Kashyap and Amit P. Sheth. Schematic and semantic similarities between database objects: A context-based approach. *Very Large Data Bases (VLDB) Journal*, 5(4):276–304, 1996.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Process and Techniques.* John Wiley & Sons, 1998.

[Lad05] Ramnivas Laddad. AOP at work: AOP and metadata: A perfect match, part 1. Technical report, DeveloperWorks, 2005.

[LB02] Andreas Leicher and Felix Bübl. External requirements validation for component-based systems. In A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Ozsu, editors, 14^{th} *Conference on Advanced Information Systems Engineering (CAiSE '02), Toronto, Canada*, volume LNCS 2348, pages 404 – 419, Berlin, May 2002. Springer.

[LBBK03] Andreas Leicher, Alexander Bilke, Felix Bübl, and E. Ulrich Kriegel. Integrating container services with pluggable system extensions. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, 5^{th} *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[Len03] Camara Lenuseni. Design Critics fr Modelle komponentenbasierter Systeme. Diploma Thesis, Technical University Berlin, Germany, June 2003.

[MB99] Mathew L. Staples Millivision and James M. Bieman. 3-d visualization of software structure. In M. Zelkovitz, editor, *Advances in Computers*. Academic Press, London, 1999.

[McC87] John McCarthy. Generality in artificial intelligence. *Communications of the ACM*, 30(12):1030–1035, 1987.

[McT93] M. F. McTear. User modelling for adaptive computer systems: A survey of recent developments. *AI Review*, 7:157–184, 1993.

[MD00] Tom Mens and Theo D'Hondt. Automating support for software evolution in UML. *Automated Software Engineering*, 7(1):39–59, 2000.

[MESW01] Bob Moore, Ed Ellesson, John Strassner, and Andrea Westerinen. Policy core information model - version 1 specification (rfc 3060). Technical report, The Internet Society, 2001.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[MRW77] J. A. McCall, P.K. Richards, and Gene F. Walters. Factors in software quality. Technical Report RADC-TR-77-369, Rome Air Development Center, November 1977.

[Mun97] Alberto Munoz. Compound key word generation from document databases using a hierarchical clustering art model. *Intelligent Data Analysis*, 1(1), 1997.

[NE00] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proc. of International Conference on Software Engineering (ICSE), Limerick, Ireland*. ACM Press, June 2000.

[NEF01] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Static consistency checking for distributed specifications. In *International Conference on Automated Software Engineering (ASE)*, Coronado Bay, CA, 2001.

[NM03] Gabor Nagypal and Boris Motik. A fuzzy model for representing uncertain, subjective and vague temporal knowledge in ontologies. In Robert Meersman, Zahir Tari, and Douglas C. Schmidt, editors, *International Conference on Ontologies, Databases, and Application of Semantics (ODBASE), Catania, Sicily (Italy)*, volume 2888 of *LNCS*. Springer, November 2003.

[oEE90] Industry of Electrical and Electronics Engineers. Ieee standard glossary of software engineering terminology. IEEE Std.610.12-1990, The Institute of Electrical and Electronic Engineers, New York, 1990.

[OMG99] OMG. UML specification v1.3. OMG-Document ad/99-06-08, 1999.

[OMG03a] OMG. UML 1.5, formal/03-03-01, March 2003.

[OMG03b] OMG. UML 2.0 infrastructure specification, ptc/03-09-15, September 2003.

[Pal00] James D. Palmer. Traceability. In Richard H. Thayer and Merlin Dorfman, editors, *Software Requirements Engineering*, pages 412–422. IEEE Computer Society, 2000.

[Pal02] Joanna Palac. Component specification. Technical Report 02-09, School of Information Technology, 2002.

[Par94] D. L. Parnas. Software aging. In *In Proc. of the 16th International Conference on Software Engineering (ICSM'94), Sorrento, Italy*, 1994.

[Pic00] Joseph P. Pickett, editor. *The American Heritage Dictionary of the English Language.* Boston: Houghton Mifflin Company, 2000.

[Pin00] Francisco A. C. Pinheiro. Formal and informal aspects of requirements tracing. In *Proceedings of the third workshop on Requirements Engineering, Rio de Janeiro, Brazil*, 2000.

[QV94] J.-P. Queille and J.-F. Voidrot. The impact analysis task in software maintenance: A model and a case study. In *International Conference on Software Maintenance (ICSM '94), Victoria, B.C.*, pages 234–242, september 1994.

[Rat04] Frank Ratzlow. Einsatzmoeglichkeiten der Aspekt-orientierten Programmierung im Kontext der Java 2 Enterprise Architektur. Diploma Thesis, FHTW Berlin, Germany, Prof. Ingo Classen, 2004.

[RE96] Matthias Radestock and Susan Eisenbach. Semantics of a higher-order coordination language. In *Coordination 96*, 1996.

[RG00] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000).* Springer, Berlin, LNCS, 2000.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models of requirements traceability. *Transactions on Software Engineering*, 27(1):58–93, 2001.

[RR98] Jason E. Robbins and David F. Redmiles. Software architecture critics in the argo design environment. *Knowledge-Based Systems. Special issue: The Best of IUI'98*, 5(1):47–60, 1998.

[RR99] Suzanne Robertson and James Robertson. *Mastering the Requirements Process.* Addison-Wesley, 1999.

[Rum98] Bernhard Rumpe. A note on semantics (with an emphasis on uml), proc. of second ECOOP workshop on precise behavioral semantics. Technical Report Technical Report TUM-I9813, Technical University Munich, June 1998.

[RV97] Paul Resnik and Hal R. Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[SdV01] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In Riccardo Focardi and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design, Tutorial Lectures [revised* *versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design, FOSAD 2000, Bertinoro, Italy, September 2000]*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer, 2001.

[SG89] Amit P. Sheth and Sunit K. Gala. Attribute relationships: An impediment in automating schema integration. In *Proc. of the Workshop on Heterogeneous Database Systems (Chicago, Ill., USA)*, December 1989.

[SG95] Standish-Group. The chaos report, 1995.

[SHU04] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Modeling pointcuts. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.

[SK92] Irene Stadnyk and Robert Kass. Modeling users' interests in information filters. *Communications of the ACM*, 35(12):49–50, 1992.

[Ski01] Martin Skinner. Enhancing a UML editor by context-based constraints for components. Diploma Thesis, Technical University Berlin, Germany, October 2001.

[SKK01] Ingo Schwab, Alfred Kobsa, and Ivan Koychev. Learning user interests through positive examples using content analysis and collaborative filtering. draft from Fraunhofer Institute for Applied Information Technology, Germany, 2001.

[Slo94] Morris S. Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994.

[SLX01] Gary N. Stone, Bert Lundy, and Geoffrey Xie. Network policy languages: A survey and a new approach. *IEEE Network*, 15(1):10–21, January 2001.

[Som92] Ian Sommerville. *Software Engineering.* Addison-Wesley, 1992.

[SP96] Clemens Szyperski and Cuno Pfister. Component-oriented programming: Wcop'96 workshop report. *Special issues in object-oriented programming: Workshop reader of the 10th European Conference on Object-Oriented Programming ECOOP'96, Linz*, pages 127–130, 1996.

[SP02] Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and .net. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 241–248, 2002.

[Spi88] J. Michael Spivey. *Understanding Z.* Cambridge University Press, 1988.

[SSR92] Edward Sciore, Michael Siegel, and Arnon Rosenthal. Context interchange using meta-attributes. In *Proc. of the 1st International Conference on Information and Knowledge Management*, pages 377–386, 1992.

[SSR94] Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using semantic values to falilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems (TODS)*, 19(2):254–290, 1994.

[ST94] Morris Sloman and Kevin P. Twidle. Domains: A framework for structuring management policy. In Morris Sloman, editor, *Chapter 16 in Network and Distributed Systems Management*, pages 433–453, 1994.

[Ste01] Lukas Steiger. Recovering the evolution of object oriented software systems using a flexible query engine. Diploma Thesis, Philosophisch-naturwissenschaftlichen Fakultät der Universität Bern, Switzerland, October 2001.

[SvdH02] Jeff Sutherland and Willem-Jan van den Heuvel. Enterprise application integration and complex adaptive systems. *Communications of the ACM*, 45(10):59–64, 2002.

[Szy97] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, Reading, 1997.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. Degrees of separation: Multi-dimensional separation of concerns. In *Int. Conference on Software Engineering (ICSE)*, pages 107–119, 1999.

[vA03] Ute von Angern. Erweiterung des Open Source Tools ArgoUML um Konzepte zum Entwurf verteilter Datenbanken. Diploma Thesis, Technical University Berlin, Germany, June 2003.

[vL01] Axel 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.

[Wan02] Jianxin Wang. Prototypische Entwicklung eines Compilers zur Umwandlung von Context-Based Constraints in ECA-Regeln. Diploma Thesis, Technical University Berlin, Germany, Research Group CIS, June 2002.

[WD93] Jennifer Widom and Umeshwar Dayal. *A Guide To Active Databases.* Morgan-Kaufmann, 1993.

[WK99] Jos B. Warmer and Anneke G. Kleppe. *Object Constraint Language – Precise modeling with UML.* Addison-Wesley, Reading, 1999.

[Zav97] Pamela Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, 1997.

[ZKS00] J. Leon Zhao, Akhil Kumar, and Edward A. Stohr. A dynamic grouping technique for distributing codified-knowledge in large organizations. In 10th *Workshop on Information Technology and Systems, Brisbane, Australia*, December 2000.