The Context-Based Constraint Language CCL for Components

Felix Bübl
Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
fbuebl@cocons.org

Technical Report of the Technical University of Berlin
Institute for Software Engineering and Theoretical Computer Science
Faculty IV Electrical Engineering and Computer Science
Report Nr. 2002-20
ISSN 1436 - 9915
Available at www.CoCons.org

17th October 2002

Abstract

Software evolution is a major challenge to software development. When adapting a systems to new, altered or deleted requirements, existing requirements should not unintentionally be violated. One requirement can affect several possibly unassociated components that do not invoke each other directly or do not even run on the same platform. A *context-based constraint* (CoCon) can indirectly select the constrained components via their metadata even if they are not associated or do not invoke each other. This paper introduces the **C***ontext-based* **C***onstraint* **L***anguage CCL*. It consists of 21 different types of CoCons for defining requirements within the logical architecture of a component-based system. A component-based system can be checked for compliance with CCL statements during (re-)design, during (re-)configuration and at runtime. This paper focuses on verifying CCL during design. The prototypical 'CCL plugin ' for the open source CASE tool ArgoUML demonstrates how to specify requirements in UML models via CCL.

Notation: In this document **new terms** are highlighted boldly where they are explained. These explanations can be found via the index at the end of this document.

Contents

List of Figures

1. Introduction

1.1 Continuous Software Engineering of Component-based Systems

1.1.1 Design for Change

Continuous Software Engineering

The context for which a software system was designed changes continuously throughout its lifetime. **Continuous software engineering** is a paradigm discussed in [62] to keep track of the ongoing changes and to adapt legacy systems to altered requirements. The system must be prepared for adding, removing or changing requirements - it must be designed for change.

Focus: Components

Improving the adaptability of software systems is a key aspect addressed in the KONTENG[1] project. Only component-based systems are addressed in this project, since this rearrangeable software architecture facilitates continuous software engineering.

Goal: Consistency

New methods and techniques are required to ensure **consistent modification steps** in order to safely transform the system from one state of evolution to the next without the unintentional violation of existing dependencies or invariants. This paper focuses on recording requirements via constraints in order to protect them from unwanted modifications. However, a new notion of 'constraint', introduced in section 2.2, is used for this approach.

Downloadable Tool

In winter term 2001/2002 a 'CCL plugin' for the open source CASE tool ArgoUML was implemented at the Technical University Berlin. The prototypical plugin is available for download at `ccl-plugin.berlios.de`. Some of the issues not discussed here are answered by this 'proof of concept' implementation, while others are up to future research.

1.1.2 Focus: Requirements Specification during Modelling

Applied in Models

A requirements specification technique is proposesd here that can be considered in different levels of the software development process: some requirements should be reflected in models, some during coding, some during deployment and some at runtime. This paper focuses on specifying requirements in models. Thus, this paper discusses how to write down which *model elements* are affected by a requirement. Obviously, there are no *model* elements during configuration or at runtime. When applying the new approach discussed here during configuration or at runtime, please read 'element' as *component* throughout the paper.

Requirements Engineering

As summarized in [43], it has long since been established that requirements management needs to be done throughout a system's lifetime. Moreover, there has been a large body of work published (e.g. [28]) on the traceability of requirements. Furthermore, goal-oriented requirements engineering is well established ([58]). The new approach presented here does not aim to replace these existing concepts. Instead, it is an addition to them.

[1] This work was supported by the German Federal Ministry of Education and Research as part of the research project KONTENG (Kontinuierliches Engineering für evolutionäre IuK-Infrastrukturen) under grant 01 IS 901 C

1.2 This Paper in Brief

CoCons

Context-Based Constraints (CoCons) have been introduced in [11]. Their basic idea can be explained in just a few sentences:

1. Yellow sticky notes are stuck onto the system's elements. They are called 'context properties' because formatted metadata describing their element's context is written on them.
2. A new constraint mechanism is used here that refers to this metadata in order to identify those elements of the system to which the constraint applies. Only those elements whose metadata fits the constraint's 'context condition' must fulfil the constraint. Up to now, no constraint technique exists that indirectly selects the constrained elements according to their metadata.
3. One constraint of this new constraint mechanism refers to *two* sets of elements. It relates each element of one set to each element of the other set. In the examples given below, one set contains all elements having a yellow sticky note, while the other set contains all elements having a blue sticky note.

Which Requirements?

This paper explains how to handle the following requirements engineering issues via the new constraint technique:

- Security requirements can demand that yellow elements must (not) access blue elements as discussed in section 4.1.
- Each time when a yellow component invokes a blue component additional services can process this communication call as examined in section 4.2.
- Distribution requirements can state that the yellow components must be allocated to the blue computers as explained in section 4.3.
- Information-Need can be expressed by writing down that the yellow users must be informed of the blue documents as described in section 4.4.
- Consistency of metadata can be improved by enforcing that elements having a yellow sticky note must not have a blue sticky note as explained in section 4.5.

Structure of this Paper

This paper consists of three parts:

1. The new constraint technique 'CoCons' is described in chapter 2. As explained above, it uses metadata. This metadata must conform to the format defined in section 2.1. A CoCon refers to this formatted metadata as described in section 2.2
2. In order to apply a CoCon to component-based systems, they must be enriched with formatted metadata as suggested in chapter 3
3. The context-based **C**ontext-Based **C**onstraint **L**anguage **CCL** consists of 21 different types of CoCons for defining requirements within the logical architecture of a component-based system. CCL is introduced in chapter 4.

Quick Tour $\xrightarrow{goto}$ Section 2

Fasten your seatbelts!If you are not interested in all the details, follow the 'Quick Tour $\xrightarrow{goto}$ (next section number)' margin labels.

2. The New Constraint Technique 'CoCons'

Overview — As sketched in section 1.2, a new constraint technique is presented here that selects its constrained elements via their metadata. This metadata must be formatted in order to use it for the new constraint techniques. First, section 2.1 proposes a format for metadata. Then, the new constraint technique that refers to this formatted metadata is explained in section 2.2.

2.1 Introducing Context Properties

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

2.1.1 What is Context?

Context is a poorly used source of information in our computing environments. As a result, we have a weak understanding of what context is and how it can be used.

Def. Context — A definition of **context** is given in [20]:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between an user and an application, including the user and applications themselves.

Instead of 'entities', the context of 'elements' is discussed here. The context of an element, like a component or an user, can be expressed as metadata.

Def. Metadata — 'Metadata' is typically defined as 'data about data'([47]). Over the last decade, metadata concepts are increasingly used to handle the required flexibility of global software and information infrastructures (see [8]). The explicit introduction of metadata concepts for many purposes results in a wide variety of specific definitions of metadata discussed, e.g., in [14, 34].

Syntax & Semantic — Semantics is about meaning; syntax is about form. Agreements about both are necessary to understand metadata in order to achieve automatic handling of metadata. On the one hand, there must be an agreed syntax (or format) for the metadata so that it can be automatically decoded. According to [2], the attribute-value pair model is the commonly used format for defining metadata today. Section 2.1.2 explains the syntax how to express context in attribute-value format. On the other hand, the meaning of the various used terms must be explained in order to enable others to interpret the metadata correctly. Useful terms for describing context are listed in chapter 3.

2.1.2 Context Properties: Formatted Metadata Describing Elements

Format — In this approach, the context of an element is expressed via metadata formatted as name and value(s). A **context property** consists of a name and a set of values as explained next and illustrated in figure 2.1.

Context Property Name

A context property **name** groups contexts of one category. For example, the names of the workflows performed by the system belong to the context property *named* 'Workflow'. Instead of 'name', 'category', 'key' or 'attribute' are terms used commonly. Only the term 'name' is used here. For each context property name, the allowed context property values must be defined.

Context Property Values

One set of **values** can be assigned to a single element e for each context property name cp. These values describe how or where this element is used – this metadata shows the context of this element. The name of the context property stays the same when assigning its values to several elements, while its values might vary for each element. For example, the values of the context property named 'Workflow' reflect in which workflows the associated element is used, as discussed in section 3.2.

Value 'None'

A special context property value exists: '**None**' represents the empty set '{}'.

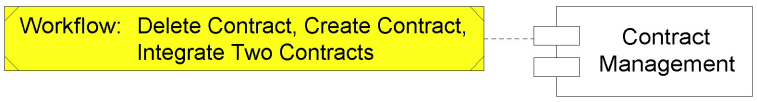

Figure 2.1: Graphical Notation for Context Properties

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 assigning values of one context property to the element e, the following syntax (format) is used on a yellow sticky note attached to e:

Syntax for Assigning Context Property Values to an Element

DirectConPropValues	::=	ContextPropertyName ['(' ElementName')'] ':' ContextPropertyValue {',' ContextPropertyValue}

Textual Example

The context property values that are assigned to the element e are separated via commas and written behind the name of the context property. For instance, "`Workflow(ContractManagement): 'Delete Contract', 'Create Contract', 'Integrate Two Contracts'`" is a valid syntax to associate the three values 'Delete Contract', 'Create Contract' and 'Integrate Two Contracts' of the context property 'Workflow' with the element e = 'ContractManagement'. Defining the associated element e in round brackets after the context property name is not required if the statement is associated with e as depicted in figure 2.1.

Graphical Notation

In figure 2.1, three values of the context property 'Workflow' are associated with the component 'Contract Management'. These values describe, in which workflows the component Contract Management is needed. 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 specified for this model element. However, it is also possible to use one context property symbol for each context property that is assigned to the same model element.

Quick Tour $\xrightarrow{goto}$ Section 2.2

The primary benefit of enriching elements with context properties is revealed in section 2.2, where they are used to specify 'context-based constraints'. Readers in a hurry can skip the next sections and proceed in section 2.2 on page 19.

2.1.3 Formal Definition of Context Properties

This section explains how the values of a context property are assigned to an element.

Def. Context Property

The formal definition of a context property as 2-tupel (cp, VV^{cp}) is introduced in [9] 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 four values allowed for cp_1 can be $VV^{cp_1} = \{$ 'New Contract', 'Delete Contract', 'Integrate Two Contracts', 'Split One Contract'$\}$. They are called valid values because only values that are contained in VV^{cp} can be associated with an element.
4. E is the set of all elements in the system (model), like e_1 = the component *'ContractManagement'*.
5. Multiple values $v_{1...n} \in VV^{cp}$ can be *directly associated* with one element $e \in E$ via the directed and transitive **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'$\}$).
6. Optionally, '**inter-value constraints**' can be defined for a context property cp. One inter-value constraint forbids or enforces values to be contained in $directvalues_{cp}(e)$. An example is discussed in section 4.5.6.

Example

The textual syntax for associating context property values with an element is defined in section 2.1.2. Values of the context property cp = 'Workflow' can be associated with the element e = 'ContractManagement' in two notations meaning the same:

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

2.1.4 Type-Instance Constraint On Context Property Values

Type-Instance Correspondence

A major purpose of modelling is to prepare one generic description that describes many specific items. This is often known as the type-instance dichotomy. According the [45], many or most of the modelling concepts in UML have this dual character, usually modelled by two paired modelling

elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs of **type-instance correspondences** include: Class-Object, Association-Link, Parameter-Value, Operation-Invocation, and so on.

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. This section discusses the impact of a context property value associated with a type on the instances of this type (according of the OMG definition of the type-instance correspondence cited above). The definition of valid values (VV^{cp} - see section 2.1.3) applies to the context property values of *all* elements on *all* levels of abstraction: for one context property name *cp* only the values contained in its set of valid values VV^{cp} can be associated with elements of the system.

Type-Instance Constraint on Context Property Values

This section defines the **type-instance constraint on context property values**. It 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 $e_{instance}$ according to this type-instance constraint on context property values. 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}.

Development Levels

Context properties can be applied in different levels of software development, like analysis, modelling, implementation, and configuration or even at runtime. Many approaches for structuring and naming these development levels exist. This section does not discuss them. Instead, it points out that a subsequent development level can deal with *instances* of the elements in the previous development level. Hence, the type-instance constraint on context property values can apply across development levels. The model of a component-based system typically specifies component types. At runtime, only instances of these component types in the model exist: $values_{cp}(e_{runtime}) \subseteq values_{cp}(e_{model-type})$

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 only assigns 5 of the 10 valid values to the component type e_{model} = 'Customer Management' via $values_{workflow}(e_{model})$ = {A, B, C, D, E} then it is not allowed to associated the instance $e_{runtime}$ of this component type with F, G, H, I, or J. At runtime, the current value of $e_{runtime}$ must be a subset of the type's values: $values_{workflow}(e_{runtime}) \subseteq$ {A, B, C, D, E} .

Current Context

If a context property values v is associated with the element e then this represents the **current context** of e at this software development level. In succeeding development levels, instances of e will exist whose context will be a subset of the $values_{workflow}(e)$= $\{v_1, v_2\}$.

Besides the type-instance constraint for context property value, another dependency between context property values of different elements can

exist that is described in the next section.

2.1.5 Belongs-To Relations Result in Derived Context Properties Values

This section explains the difference between *directly associated* and *derived* context property values.

Do I Need This? As soon as context properties are associated with *different* elements that *belong to* each other the mechanism introduced in this section facilitates managing the elements metadata. 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 only once to the computer in order to assign it to all components deployed to this computer automatically. If you prefer to attach the same yellow sticky note 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. Then again, 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** $derivedvalues_{dcp} : E \rightarrow \mathcal{P}^{VV^{dcp}}$:

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

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:

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

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

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

'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 assigned to e via the $derivedvalues_{cp}(e)$ mapping from every element e_i to which e belongs. When implementing a context-property-aware tool, like a modelling tool, only the *directly associated* values must be made persistent because the derived values can be obtained from the associated values. The $derivedvalues_{cp}(e)$ should be displayed in italics in diagrams as demonstrated in figure 3.3.

'*cp* contains v'

In this paper, often the term 'for the element e, the context property *cp* contains the value v is used. It means that $v \in values_{cp}(e)$. The term 'for e, the context property *cp* **has** v' is equivalent to '$v \in values_{cp}(e)$'. It is not equivalent to 'v is directly associated with e' because this term means $v \in directvalues_{cp}(e)$.

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

As explained in section 2.1.4 both types and instances of a type can exist in a system model. 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

In section 3.8, belongs-to relations are **explicitly** defined on the metamodel level only. The metamodel contains *types* of model elements. The instance of a metatype is a model element. Hence, *implicit* belongs-to relations for the diagrams of the UML components approach are defined in section 3.8 by specifying explicit belongs-to relations between meta*types*. The benefit of only explicitly specifying belongs-to relations in the metamodel is that during design or in later development levels, only implicit belongs-to relations are used. This is less confusing. However, it is useless to define a belongs-to relation in the metamodel whose belongs-to criteria cannot be automatically decided. An example is discussed in section 3.8.

Pitfalls

Belongs-To relations are not easy to understand. Somehow, every element of the system belongs to any other element. A belongs-to relation between two elements should only be defined if the context property values shall be derived from one element to the other one. Hence, it takes some effort to define them properly.

Benefit: Belongs-To Hierarchy

The consequence of defining a belongs-to relation is that a context property value assigned to 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 assigned to 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. An example for using belongs-to relations is given in section 3.8.

2.1.6 System Properties

Queried From The Middleware Platform

Usually, the current context property values associated with a component are defined manually. On the contrary, **system properties**, like 'the current user' or 'the current IP address', can be automatically queried from the middleware platform during configuration or at runtime. Each middleware technology enables certain system properties to be queried, but it's beyond the scope of this paper how the various technologies query the system property values. This paper focuses on *manually defined* context properties that are not automatically available due to the underlying middleware platform. However, section 2.1.9 and section 2.2.3 suggest to consider system properties already in models.

2.1.7 Dependent Context Property Values

Motivation

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

'Unused' Values

A dependent value assigned to an element is **unused** if it does not comply with its dependency. 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.

The Syntax for Dependent Context Property Values

ContextPropertyValue	::=	OneOrMoreValues ['(' ('**in state**' NameOfState {',' NameOfState }*) | ('**if**' ContextCondition {'**OR | AND** ' ContextCondition }*) | Constraint) '**)**']
OneOrMoreValues	::=	Value | ('**{**' (Value)+^{Comma} '**}**'

Syntax

If a context property value depends on something, this dependency is written down in round brackets behind the context property value(s). If several context property values associated with the same element depend on the same, they are grouped within curly brackets. Different types of dependencies are introduced now via examples for the element e:

State-Dependent

"`CurrentWorflow(e): 'New Customer'(in state` s_4`,`s_5`), {'Delete Customer', 'Modify Customer'}(in state` s_1`), 'Sell Product'`" represents **state-dependent** values if s_i are names of well-defined states. Other specification techniques, like state diagrams or petri nets, are

needed to define these states. At runtime, the current value(s) depend on this state. If a context property value is specified without a given state, like 'Sell Product' in this example, it applies to *all* states of e.

Constraint-Dependent

"`CurrentWorflow(e): 'New Customer'(if self.age` $<$ `18), {'Delete Customer', 'Modify Customer'}(if self.age` $\geq$ `18)` "represents **constraint-dependent** context property values. The constraint within the round brackets is typically specified in the Object Constraint Language OCL in UML models. An OCL constraints refers to the *state* of instances of the associated element as explained in section 5.2.2. Hence, a constraint-dependent value is similar to a state-dependent value. The only difference is that state-dependent values depend on the (abstract) name of a state, while constraint-dependent values depend on a constraint that precisely specifies a state.

Formula-Dependent

"`CurrentWorflow(e): ?(= self.GetWorkflowname()), 'Modify Customer'`"illustrates **formula-dependent** context property values. The formula within the round brackets always starts with '=' (as in Microsoft Excel) and refers to the attributes or methods of the associated element e or other elements that are associated with e. The formula defines how to calculate the value. In the example given, the value of 'CurrentWorkflow' is extracted by invoking the method 'GetWorkflowname()' of e that returns the current value of the context property 'CurrentWorkflow'. Calculating the context properties value from the values of the attributes or the methods is a topic of future research, though. Formula-Dependent values are similar to constraint-dependent values. Again, the formula refers to the *state* of instances of the associated element(s). In contrast to constraint-dependent values, the result of the formula is not pre-defined. In the example given in the previous paragraph, the value '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.

Context-Condition-Dependent

Context conditions are introduced in section 2.2.1. In "`CurrentWorkflow(e): {'Delete Customer', 'New Customer'(if System.UserRole EQUALS 'Controller'), 'Modify Customer'}(if System.UserRole EQUALS 'Trader'), 'Sell Product'`"the value 'New Customer' represents a **context-condition-dependent** context property value. This kind of value dependency can also be expressed via a value-binding CoCon introduced in section 4.5. The association of an element to a context property value depending on a context condition is only *in use* if the context condition matches to the other context property values assigned to the same element. If the context property on which a context-condition-dependent value v depends is a system property then v is called **system-dependent** context property value.

Naming Convention

The name of a context property with dependent values should emphasize this dependency. It is recommended to start the name of state-dependent context property with 'Current'.

No Recursion

No recursion is allowed. A dependent context property value must not dependent on another dependent context property value for two reasons. On the one hand, the comprehensibility might get out of hand. On the other hand, unrestricted recursion would inhibit calculability of context conditions. Future research might find a solution for allowing recursion, though.

Dependent context properties are not deeply examined here and will be topic of future research. Nonetheless, the effectiveness of dependent con-

text property values is demonstrated in the next section.

2.1.8 Coping with Contexts for Different Configurations

Configurations

Different values of the same context property can be assigned to 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 2.2 illustrates how to model two different configurations 'BSH' (Building Society Schwäbisch Hall) and 'TUB' (Technical University Berlin) via the context property 'Configuration'.

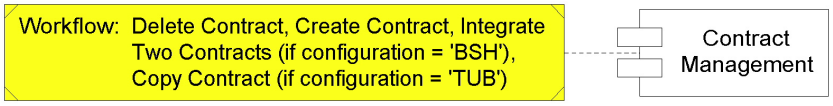

Figure 2.2: Context Property Values Can Depend On The Current Configuration

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 assigned to the model element e ='Contract Management' as illustrated in figure 2.2. 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.

Coping with Different Versions or Variants

As well as different configurations, different versions or different variants of the same element can be considered via dependent context property values. Again, a context property 'Version' describing the version and its valid values are defined. Then, for an element the values of other context properties can depend on the value of 'Version'.

2.1.9 Useful Context Property Stereotypes

Two useful stereotypes for refining the semantic of a context property are suggested:

≪Number≫ : The values of a ≪Number≫ property are numbers. For example, they can describe 'quality of service' information and, thus, facilitate establishing reasonable load balance as described in [12] and in section 4.3.4. In addition to normal context properties, the particular measurement is added in round brackets to the context property name, e.g. '≪Number≫ RPCs (calls per Day)' — RPC is an abbreviation of Remote Procedure Call . ≪Number≫ context properties are typically not derivable (see section 2.1.5)

≪System≫ : ≪System≫ properties are introduced in section 2.1.6. They indicate metadata that can be queried from the system during configuration or at runtime. When used in a model, a system property defines the range of allowed values as explained section 2.1.4. By setting (may

be only one) value(s) the system property defines in which way the associated model element will be or has been implemented. For example, the context property ≪System≫ TransactionMode' (see section 3.4) can have the value 'Requires a Transaction' for a certain component already in the model. When querying the current value of '≪System≫ TransactionMode' for this component from the system during configuration or at runtime, it must be one of the values defined in the model. Hence, it is not allowed to configure a transaction mode for this component that has not been specified as system property value in the model. System properties facilitate to consider important implementation aspects already in models as suggested in [12]. Referring to technology leads to frequent changes of the system property values because of the fast evolving IT market. Hence, rather abstract values should be given.

2.1.10 Navigation via Prefix-Dot-Notation

Adding Prefixes to Context Property Names

As defined in section 2.1.2 and 2.1.3, a context property consists of a *name*, like 'workflow', and values. In the previous sections, the association of context property values with (model) elements has been discussed. This sections explains how to refer to a context property value that has been associated with another element if the other element is associated with the focussed element. Additionally, this section discusses the association of context property values with metatypes in a metamodel.

Associating Context Property Values with Metatypes

A metamodel consists of metatypes (also called meta classes in UML). If context property values are associated with a metatype then the instances of this metatype (the model elements) cannot be associated with any value of this context property that is not associated with their metatype according to the type-instance constraint on context property values (see section 2.1.4). Moreover, if a context property is associated with a metatype then this association can be navigated later on as explained next.

Syntax

In order to reference context property values of other elements the path to the other element is added to the context property name as a prefix:

The Syntax of the Prefix-Dot-Notation

ContextPropertyName ::=	[(Elementname | Rolename) {'**.**' (Elementname | Rolename) }*] ContextPropertyName

Navigation

As in OCL ([60]) dot-notation is used to navigate along associations here:

- The association of a context property value with an element of one metatype can be referenced by writing down a dot between the name of the metatype and the name of the context property. The result of a writing down the name of a metatype as a prefix before the name of a context property is that only those values of this context property that are associated with instances of this metatype are taken into account as discussed in section 2.2.3.

  For instance, the context property 'Location' describes the position of elements. The elements can be instances of different metatypes, such as 'User' or 'Component'. When referring to the 'Location', it may be better to clearly state if an user's location or a component's location is addressed via 'User.Location' or 'Component.Location'. Are more detailed example is given in section 4.4.5.

- It may be necessary to refer to a context property value that is associated with another element than the focussed one. 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 via the dot-notation. Navigating via the dot-notation from one element to another is thoroughly explained in [60]. 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 as discussed in section 2.2.3.

  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 'Product.Contract.Workflow' refers to the value 'Customer Marriage' associated with Contract. Again, are more detailed example is given in section 4.4.5.

2.1.11 Research Related To Context Properties

.NET/(D)COM(+) Many techniques for writing down metadata exist The notion of context or container properties is well established in component runtime infrastructures such as COM+, EJB, or .NET. The new concept and benefit of enriching elements with context properties is revealed in section 2.2, where they are used to select constrained elements.

Tagged Values Context properties are similar to tagged values in UML - on the design level, tagged values can be used to express context properties. In contrast to tagged values, the values of a context property must fulfil some additional semantical constraints. For example, the values of one context property for one model element are not allowed to contradict each other (see inter-value constraints example in section 2.1.3 and section 4.5.6). Furthermore, not every value is allowed in a context property, but only valid values (VV^{cp} - see section 2.1.3). Moreover, dependencies between context property values can exist as discussed in section 2.1.7. Furthermore, the values of a context property associated with an element can be derived from another element via a belongs-to relation as explained in section 2.1.5 and in section 3.8. 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. Existing grouping mechanisms like *inheritance*, *stereotypes* ([5]) or *packages* are not used because the values of a context property associated with one model element might vary in different configurations or even change at runtime. Typically, multiple values of several context properties are assigned to the same model element e. Let a system have n different valid values in $VV^{Workflow}$. Then for each $v_i \in VV^{Workflow}$ a model element e 'is' or 'is not' (= 2 possibilities) assigned to v_i. Allowing for all possible combinations, you would need up to 2^n different stereotypes in UML 1.3 because according to section 2.6.2.2 in [44], it is not allowed to have more than one stereotype per model element. It is illustrated in the diagram, p. 2-70 of [44], which specifies 0..1 for stereotype multiplicity in relation to the extendedElement.

...Cannot Change at Runtime

This multiplicity has changed to $0..*$ in UML 1.4. So that in UML 1.4 only n different stereotypes are needed for assigning any of n valid values of the context property 'Workflow' to it. However, even UML 1.4 multi-inheritance of stereotypes is unsuitable, since usually the values of more than one context property are assigned to 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 (see section 2.1.8) or change at runtime (see section 2.1.7). A model element is not supposed to change its stereotype or its package at runtime. One context property can be assigned to different types of model elements. For example, the values of 'Workflow' can be associated both with 'classes' in a class diagram and with 'components' in a component diagram. Using packages or inheritance is not as flexible. According to [45], stereotypes can group model elements of different types via the baseClass attribute, too. However, this 'feature' has to be used carefully and the instances of a model element are not allowed to change their stereotype at runtime. Context properties are a simple mechanism for grouping otherwise possibly unassociated model elements - even across different views, diagram types, specification techniques, systems or platforms.

2.2 Introducing Context-Based Constraints (CoCons)

This section presents a new constraint technique called 'CoCons' for requirements specification introduced in [11]. The language CCL introduced here consists of CoCons for component-based systems.

2.2.1 New: Indirect Selection of Constrained Elements

CoCons

One requirement can affect several possibly unassociated elements. During design, the affected model elements may not be associated with each other or even may belong to different models. At runtime, the affected elements may not invoke each other directly or do not even run on the same platform. A *context-based constraint* (CoCon) can indirectly select the constrained elements via their metadata. If the context property values of an element comply with the CoCon's context condition then the CoCon applies to this element. The simplified metamodel in figure 2.3 shows the abstract syntax for CoCons. The metaclasses 'ModelElement' and 'Constraint' of the UML 1.4 'core' package used in figure 2.3 are explained in [45].

Evolution Needs Invariants

CoCons should be preserved and considered in model modifications, during deployment, at runtime and when specifying another – possibly contradictory – CoCon. Thus, a CoCon is an *invariant*. If a requirement is written down via a CoCon, its violation can be detected *automatically* as described in chapter 4.

Example A

If values of the context property 'Workflow' are assigned to elements as suggested in section 2.1.2 then a CoCon can state that "*All components belonging to the workflow 'Integrate Two Contracts' must be inaccessible to the component 'Customer Management'*" **(Example A)**. This constraint is based on the context 'workflow' of the components – it is a *context-based* constraint.

Example B

Another requirement might state that "*The component 'EmployeeMan-*

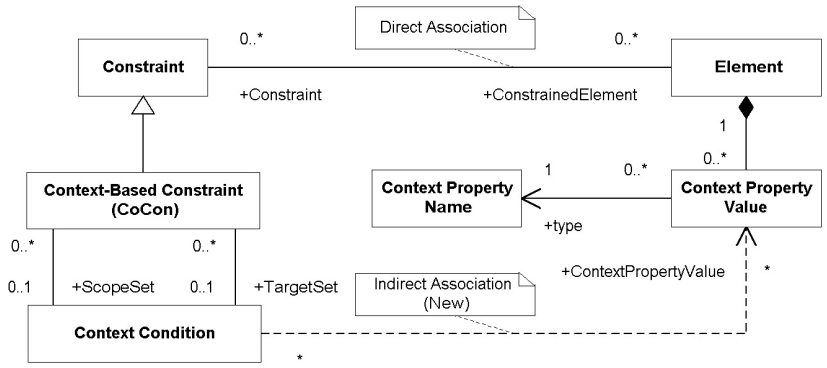

Figure 2.3: The Simplified CoCon Metamodel

agement' must be inaccessible to all components whose context property 'Operational Area' contains the value 'Field Service'" **(Example B)**.

Context Condition — One CoCon applies to the elements whose context property values fulfil the CoCon's **'context condition'**. A context condition describes a (possibly empty) *set* of elements.

Restriction — A context condition may be restricted to elements of one metaclass – in UML ([45]), this metaclass is also called 'baseClass'. Throughout this paper, the context conditions are mostly restricted to 'components'. The result of a **restriction** is that no model elements of other metatypes, e.g. 'classes', are selected even if their context property values fit the context condition.

Range — A **range** can limit the number of elements that are selected by a context condition. The range mechanism is not discussed in here – it is needed to specify 'flexible CoCons'.

Two Sets: — A CoCon always relates two sets of elements. It relates each element of one set to each element of the other set. More precisely, it relates any element of the 'target set' with any element of the 'scope set' as explained next.

Target Set — On the one hand, a context condition can determine the **'target set'** containing the elements that are checked by the CoCon. In example A, the target set is selected via the following context condition: "*All components belonging to the workflow 'Integrate Two Contracts'*".

Scope Set — On the other hand, a context condition can select the **'scope set'** that represents the *part of the system*, where the CoCon is enforced. In example A, the scope of the CoCon is a single element – the component 'Customer Management'. Nevertheless, the scope set of a CoCon can contain any number of elements, as illustrated in example B.

Selecting Set Elements... — Both target set elements and scope set elements of a CoCon can be specified either directly or indirectly:

...directly : Set elements can be specified by directly associating the CoCon with the element(s). In the textual language introduced in section 2.2.5, this direct association is defined by naming the element(s) or by using the keyword '`THIS`', like '`self`' in OCL. In example A, the CoCon is associated *directly* with the 'Customer Management' component. This unambiguously identifies the only element of this scope set.

...indirectly (new) : **Indirect association** is the key new concept of context-based constraints. Set elements can be *indirectly associated* with a CoCon by a context condition. The scope set in example B contains all the components whose context property 'Operational Area' contains the value 'Field Service'. These scope set elements are anonymous. They are not directly named or associated, but described indirectly via their context property values. If no element fulfils the context condition, the set is empty. This simply means that the CoCon does not apply to any element at all.

This 'indirect association' is represented as a dotted line in fig. 2.3 because it is not a UML association. Instead, it resembles a UML dependency that only exists if the element's context property values fit the context condition. The 'indirectly associated' elements are selected by evaluating the context condition each time when the system is checked for whether it complies with the CoCon.

Combining Element Selections In order to describe the elements of either one target set or one scope set, a combination of direct or indirect element selections can be used. Typically, several element selections are combined with 'or' conditions. For example, a target set can contain "the component 'CustomerManagement' *or* all components that are used by the field service". 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 by the field service. 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*. By replacing *or* with *and* in the example, the component 'CustomerManagement' would only be contained in the set if it is used by the field service, too. If the component 'CustomerManagement' does not fulfil both element selections combined via *and* then it is not contained in the set. Moreover, any other component used by the field service will not be contained in the set because it will only be selected via the indirect selection and doesn't fulfil the direct selection that demands its name to be 'CustomerManagement'. Usually, *or* is used for combining conditions.

Total Selection One kind of indirect association exists that doesn't refer to context property values: the **total selection** simply selects *all* elements regardless of their context. However, it can be *restricted* to select only all elements of a certain metaclass, like all 'components'. An example is given in section 4.4.5. The total selection is used to specify default CoCons as discussed in section 4.1.

'Among Whom ...Exist' Sometimes, we want to specify that there is at least one element in a set for which a certain condition holds. The 'among whom at least *x elements* exists' operation on either the target or the scope set can be used for this purpose. It can be added after a set selection and defines that the *x elements* must be contained in the selected set. The *x elements* can be specified either via direct or via indirect selection.

Simple CoCons A context-based constraint is called **simple** if either its target set or its scope set contains *all* elements of the whole system via a total selection or if it contains exactly one directly associated element.

CoCon Type In this introduction, only CoCons of `ACCESSIBLE TO` type are discussed. They specify that the target elements can (not) be accessed by the el-

ements in the CoCon's scope set. Section 2.2.7 examines how to define CoCon types, and chapter 4 lists more CoCon types.

Difference between Target Set and Scope Set?

A CoCon refers to two different set of elements: one describing *which* elements are controlled by the CoCon (contained in the target set), and one describing *where* the target elements are checked for whether or not they comply with the CoCon. Yet, in some cases the names 'target set' and 'scope set' do not seem appropriate. Mixing example A and example B, a CoCon could state that "*All components belonging to the workflow 'Integrate Two Contracts'* (Set_1) must be inaccessible to *all components whose context property 'Operational Area' contains the value 'Field Service' (*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 types (see section 4) the names 'target set' for Set_1 and 'scope set' for Set_2 fit well. Perhaps future research reveals names that always fit well.

2.2.2 Detecting Conflicting Requirements

Example

One CoCon can contradict other CoCons. Example 'A' in section 2.2.1 states: `ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Integrate Two Contracts' MUST NOT BE ACCESSIBLE TO THE COMPONENT 'Customer Management'` However, this CoCon can contradict another CoCon stating that `ALL COMPONENTS WHERE 'Personal Data' = 'True' MUST BE ACCESSIBLE TO THE COMPONENT 'Customer Management'` if any component needed in the workflow 'Integrate Two Contracts' handles personal data.

Automatically Detectable Conflicts

In order to reveal conflicting requirements and to detect problems early on, they should be written down in models. Fixing them during implementation is much more expensive. Conflicting CoCons can be automatically detected. This is a major benefit of CoCons. Details on how to detect conflicting CoCons are explained for each CoCon type family in the chapter 4.

Quick Tour $\xrightarrow{goto}$ Chapter 3

Quick readers can skip the next sections and proceed with section 3 on page 29. In BASIC: `GOTO 3`. Nevertheless, be careful - according to [21], jump instructions are "considered harmful".

2.2.3 Details on Context Conditions

System Properties

A CoCon can be specified in the model, but its context condition cannot be evaluated at the modelling level if it refers to values that are not defined yet. In this paper, a context condition usually refers to context property values that have been defined in models. However, a context condition can refer to system property values (see section 2.1.6) in models even if their values are not defined in the model. This simply means that the context condition cannot be evaluated at the modelling level – the model elements it selects cannot be identified. Hence, it cannot be checked whether the system model complies with this CoCon. It can be checked in later levels of the development process as soon as the referred system property values can be queried from the middleware platform.

Ignoring Unused Values One context condition can refer to several context property values $v_n, ..., v_m$ of possibly several context properties $cp_k, ...cp_l$. For each $cp_i (k \leq i \leq l)$ only those values of an element e ($values_{cp_i}(e)$) must match with context condition that are *in use* (see section 2.1.7).

The Condition refers... Two different kinds of context conditions exist. A context condition can 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 can compare sets. According to section 2.1.3 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 values via the following conditions: The condition 'CONTAINS' demands that one set is subset of or equal to ($\subseteq$) the other set. Moreover, five 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 can compare one value $v_{compare}$ with each value $v_i \in values_{cp}(e)$. If v_i is a string, than = and != are allowed conditions for comparing it with $v_{compare}$. If v_i is a number – as for «Number» (see section 2.1.9) context properties then — besides = and != — the usual logical conditions for comparing numbers are allowed: $>$, $\geq$, $<$, 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.

Using Other Query Languages The query language used to specify the context condition depends on the format of the metadata it refers to. The simple and flat format 'context properties' for metadata is introduced in section 2.1. Of course, more complex formats for storing the metadata of one element, like hierarchical, relational, or object-oriented schemata, can be used. If, e.g., the metadata for one element is stored in a relational schema, then the context condition can be expressed in SQL.

Considering the Prefix... When evaluating a context condition, each element is checked whether its context property values match to the context condition or not. Nevertheless, a context condition can refer to other context property values that are not associated with the currently checked element. As explained in section 2.1.10, a prefix-dot-notation can be used to refer to other context property values via prefixes added to the context property names. Two different reasons for adding prefixes to context property names used in a context condition are discussed here:

...When Referring to the 'Other' Constrained Element A CoCon relates any element of the target set to any element of the scope set. When focussing on one element of one set, a related element in the other set is called 'other' element here. A context condition for selecting a scope set element can refer to a context property value associated with the 'other' (target set) element and vice versa by addressing the 'other' element via the prefix notation. Typically, the target set and the scope set of a CoCon are restricted to elements of a metatype. This metatype is also called 'baseClass' in UML. For example, if the baseClass of a scope set is defined as 'components' then the scope set is restricted to contain nothing but components. If a set is restricted to a different baseClass

than the other set then one set's context condition can refer to a context property value associated with an 'other' element by adding the name of the baseClass as prefix to the name of this context property. An example is given in section 4.4.5.

...When Navigating Along Associations: In order to find out which elements are selected by a context condition, each element (of the restricted baseClass) must be checked if its context property values match with the context condition. As explained in section 2.1.10, prefixes added via dot-notation to the context property name can be used to navigate to a context property that is not associated with the currently checked element. Instead, it is associated with another element that is associated with the checked element via (may be several nested) associations that are described via the prefix-dot-notation. Hence, the currently checked element can be selected due to the context property value associated with another element. Again, an example is given in section 4.4.5.

Defining an Alias: As explained above, a context condition can refer to a context property value associated with the 'other' element in a context condition by prefixing the baseClass of the other set to the context property name. If the checked element is of the same baseClass as the 'other' element then an alias can be defined for at least one of the baseClasses. Once more, an example is given in section 4.4.5.

$\overset{belongs^*}{\longrightarrow}$ Closure: As explained in section 2.1.5, the belongs-to relation is transitive. The **transitive closure** $\overset{belongs^*}{\longrightarrow}$ contains all elements where v is assigned to due to a belongs-to relation. If v is directly associated with $c \in E$ and $a \overset{belongs}{\longrightarrow} b \overset{belongs}{\longrightarrow} c$ then $a \overset{belongs^*}{\longrightarrow} c$. If belongs-to relations are defined then the transitive closure $\overset{belongs^*}{\longrightarrow}$ must be considered when evaluating a context condition.

Naive Algorithm: A 'naive' algorithm for calculating all elements selected via a context condition consists of the following steps:

For each element $e \in E$

1. for each context property name *cp* addressed in the context condition
   (a) Calculate the full transitive $\overset{belongs^*}{\longrightarrow}$ closure $derivedvalues^*_{cp}(e)$. Many algorithms exist for calculating a transitive closure. A well-known one is the Floyd-Warshall algorithm. It was published by Floyd ([25]), and is based on one of Warshall's theorems ([61]). Its running time is cubed in the number of elements. Thus, all values $values_{cp}(e) = directvalues_{cp}(e) \cup derivedvalues_{cp}(e)$ of the context property name *cp* are identified.
   (b) If $values_{cp}(e)$ don't match with the context condition, than don't select this element. Otherwise, select this element.

2.2.4 Attributes of Context-Based Constraints

CoCon Attributes: 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 relationships between scope set elements and target set elements. What happens if two CoCons of the same CoCon type apply to one element e_1, but one CoCon relates e_1 to e_2, while the other CoCon does not relate e_1 to e_2? For example, element e_1 must be inaccessible to e_2 due to $CoCon_A$, and e_1 must be inaccessible to e_3 due to $CoCon_B$. $CoCon_A$ is of the same CoCon type as $CoCon_B$. If both $CoCon_A$ and $CoCon_B$ are of the same CoCon type and have the same `PRIORITY` then they both apply. If, however, their CoCon type is the same but priority differs 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.

Priority rules are not discussed in detail here, but the following suggestion is offered: 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 2.2.1. 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.

`ACTION` A CoCon relates each element in its target set with each element in the scope set. The attribute **ACTION** describes what action must be taken if two elements are related via this CoCon and comply with the CoCon. Its value depends on the development level in which the CoCon is checked. Hence, each value of the attribute `ACTION` should start with 'ALWAYS', 'DURING MODELLING', 'DURING CODING', 'DURING CONFIGURATION' or 'AT RUNTIME'. If no development level is given, 'ALWAYS' is used as default. Furthermore, it depends on the CoCon type which `ACTIONS` are useful values of this attribute. Thus, they are discussed in section 4.

`ELSE-ACTION` The attribute **ELSE-ACTION** describes what action must be taken if two elements are related via this CoCon but don't comply with the CoCon. Its value depends on the development level in which the CoCon is checked. Hence, each value of the attribute action should start with 'ALWAYS', 'DURING MODELLING', 'DURING CODING', 'DURING CONFIGURATION' or 'AT RUNTIME'.

2.2.5 The Generic CoCon Syntax

Generic Syntax This section introduces the generic syntax of context-based constraints. The syntax is generic because three of its BNF rules must be refined for each CoCon type: `CoConType`, `Restriction` and `Restrictions` depent on the application area of CoCons. Before explaining how to refine these rules in the next section, the application-domain-independent rules are listed here. The semantics of the terms used here are explained in section 2.2.

$()+^{Seperator}$ Rules concerning the separator (',', 'OR' or 'AND') are abbreviated: "`Rule { Separator Rule }*`" is abbreviated "`(Rule)`$+^{Separator}$".

Generic Syntax of Context-Based Constraints

CoCon	::=	TargetSet **'MUST'** [TypeCondition] **'BE'** CoConType ScopeSet [**'WITH'** (Attribute)$+^{AND}$]
TypeCondition	::=	**'NOT'** | **'ONLY'**
TargetSet	::=	ElementSelection
ScopeSet	::=	ElementSelection
ElementSelection	::=	(IndirectSelection | DirectSelection)$+^{(OR \mid AND)}$ [**'AMONG WHOM AT LEAST'** (IndirectSelection | DirectSelection)$+^{(OR \mid AND)}$ (**'EXIST'** | **'EXISTS'**)]
DirectSelection	::=	(**'THE'** [RestrictionAlias] Restriction ElementName) | **'THIS'**
IndirectSelection	::=	Range Restrictions [**'WHERE'** ContextCondition$+^{(AND \mid OR)}$]
Range	::=	**'ALL'** | Number | (**'BETWEEN'** LowerBoundNumber **'AND'** UpperBoundNumber **'OF ALL'**)
ContextCondition	::=	CompareTwoSets | SingleSetCondition
SingleSetCondition	::=	ContextPropertyName (**'IS EMPTY'** | **'IS NOT EMPTY'**)
CompareTwoSets	::=	ContextPropertyName CompareCondition SetOfConPropValues
SetOfConPropValues	::=	ContextPropertyValue | (**'BOTH'** ContextPropertyValue (**'AND'** ContextPropertyValue)+) | (**'EITHER'** ContextPropertyValue (**'OR'** ContextPropertyValue)+ (**'OR A COMBINATION OF THEM'** | **'BUT NOT A COMBINATION OF THEM'**) | (**'EXACTLY THE SAME VALUES AS'** ContextPropertyName) | (**'AT LEAST ONE VALUE OF'** ContextPropertyName) | (**'ONLY ONE VALUE OF'** ContextPropertyName)
CompareCondition	::=	**'CONTAINS'** | **'DOES NOT CONTAIN'** | '=' | **'DOES NOT EQUAL'** **'INTERSECTS WITH'** | **'DOES NOT INTERSECT WITH'** | '<' | '>' | '<=' | '>='
Attribute	::=	AttributeName '=' (AttributeValue)$+^{Comma}$
AttributeName	::=	**'ACTION'** | **'ELSE-ACTION'** | **'COCONNAME'** | **'COCONAUTHOR'** | **'COMMENT'** | **'PRIORITY'**

Example A The syntax and semantic of the CoConType, Restriction, and Restrictions rule are domain-specific. In chapter 4, CoCon types for component-based systems are defined. For instance, section 4.1.2 introduces the CoConType 'ACCESSIBLE TO. According to generic BNF rules given here, example 'A' in section 2.2.1 can be specified as follows:

```
ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Integrate Two Contracts'
MUST NOT BE ACCESSIBLE TO THE COMPONENT 'Customer Management'.
```

Quotation Marks — A context property name, a context property value, and an attribute value must be given in quotation marks. Either simple 'quotation marks' or double "quotation marks" are allowed.

Total Selections — Total selections are defined by omitting the '`WHERE ...`' clause in the `Indirect Selection` rule.

ALL ELEMENTS — 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 4.5.6.

THIS — The terminal symbol '`THIS`' in the `DirectSelection` rule has the same meaning as '`self`' in OCL (see [18, 60]). If the CoCon is directly associated with a model element via a UML association then '`THIS`' refers to this model element. For example, if a CoCon is associated with an interface of a component then THIS refers to this interface.

CoCon-Type-Condition — The rule `TypeCondition` allows to place a `NOT` or an `ONLY` after the keyword `MUST`. This **CoCon-type-condition** changes the semantic of a CoCon type. Thus, its effect is explained for each type in chapter 4.

2.2.6 Graphical Notation Guide to Context-Based Constraints

Visual CCL — Constraint diagrams provide a visual notation for expressing constraints. As suggested in [27], constraint diagrams should be used in conjunction with the UML as a visual technique. A symbol for context properties has been suggested in figure 2.1. This section proposes to depict a CoCon as (European) warning sign labelled 'i' for **i**nvariant. The general layout of context-based constraints is illustrated in figure 2.4. In terms of visual language research ([1]), it shows a rudimentary, attributed graph grammar that determines the layout. The warning sign symbol is attached to the target set specification. An arrow points from the warning sign to the scope set. A label on the arrow describes the CoCon type. CoCon attributes may be specified beneath the arrow.

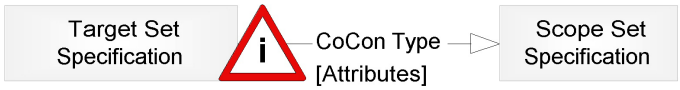

Figure 2.4: Visual Layout of Context-Based Constraints

Context Condition — A context condition uses the same graphical symbol as for context properties. The context condition syntax is introduced in section 2.2.5 via the `Context Condition` rule.

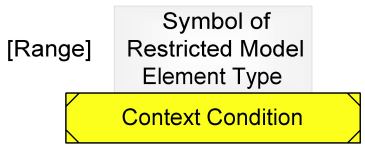

Figure 2.5: Visual Layout of Context Conditions

As explained in section 2.2.1, a context condition is usually restricted to one element type (e.g. a 'component'). Only elements of this type are checked for, whether their context property values fulfil the condition. A restriction is depicted above the condition by the type's symbol bearing a question mark instead of a name. Examples of CoCons in visual language are given in [12].

2.2.7 Defining a CoCon Type

In chapter 4, the language CCL is introduced. It consists of 21 different CoCon types. A **CoCon type definition** consist of the following details:

CoCon Type : The name of the CoCon type must be defined by adding it to the CoConType rule of section 2.2.5.

Constrained Element Types : A nonempty list of the types of elements that are allowed in the target set, respectively in the scope set.

Semantics : For each combination of allowed target set element type and allowed scope set element types (see above) the impact of this CoCon on these set elements must be defined. For instance, what happens if a class is an element of the target set, and a component is an element of the scope set?

CoCons can be applied during different development levels: they can be used in models, during configuration or at runtime. The semantics must be defined for each development level. In this paper, mostly the modelling level is discussed.

Detectable Conflicts : Detectable conflicts both between CoCons of the same and of other types of the same CoCon type family are described.

Examples are given in section 4, where CoCon types are defined that are useful for modelling component-based systems. Future research will result in additional CoCon types.

2.2.8 The Fundamental Things Apply As Time Goes By

CoCons have been invented in 2000. In the meantime, some papers 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 [9] and [10], context-based 'constraints' were called context-based 'instructions'. All later publications use the term CoCons. Different CoCon types have been proposed. Sometimes, the same CoCon type has been renamed. `NOT ACCESSIBLE TO` CoCons, for instance, have been called `InAccessibleBy` CoCons. Their semantic, however, stayed the same.

3. Enriching Component Models with Context Properties

The model of a component-based system must be enriched with context properties in order to apply context-based constraints to it. First, section 3.1 sketches different approaches for modelling component-based systems. The following sections suggest context properties useful in the 'UML components' modelling approach. When designing a particular system, some of the proposed context properties might be taken out and some unmentioned ones might be added by the developer as needed.

3.1 Modelling Component-Based Systems

Component model

The **component model** depicts the architecture of the component-based system. Besides components and their interfaces, it possibly shows connectors, and describes, which business objects are used by which interface. A component model only displays the most important details of a component-based system. It does not provide a glass box view.

ADL's / UML

To describe architectures of software systems numerous specific languages called Architecture Description Languages (ADLs) have been developed as summarized in [38]. However, some research and industrial projects are using the standard Unified Modelling Language (UML) for representing software architectures of systems. 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 methods and techniques for modelling component-based systems [15, 16, 22, 23, 33, 52, 57, 63] have been proposed.

UML Components

Many modelling or specification techniques exist for modelling component-based systems. The new constraint language CCL proposed here can be applied to all of them. This paper concentrates on how to apply it to the 'UML components' approach introduced in [16]. Adapting CCL to other approaches is subject of future research. The UML components approach has been chosen because it uses standard UML techniques, like class diagrams, to specify the component model. Moreover, UML is widely used.

CCL Plugin for ArgoUML

Various software tools for specifying models of component-based systems exist. **ArgoUML** is an open source CASE tool for designing software systems via UML. In winter term 2001/02 a 'CCL plugin' ([55]) for ArgoUML has been implemented at the TU Berlin. It is available for download at `ccl-plugin.berlios.de`. The CCL plugin adds several 'UML components' diagrams for designing component-based systems to ArgoUML. These additional diagrams are described next. Moreover, it adds support for the Context-Based Constraint Language CCL described in chapter 4. The following 'UML components' diagrams are added to ArgoUML via the CCL plugin:

- The *'Business Type Diagram'* integrates those business objects ([54]) used by several components into one single, global view. It is presented in section 3.3.

- The *'Component Specification Diagram'* describes the system's logical component structure. It is illustrated in section 3.4.
- The *'Interface Information Diagram'* specifies details for one interface as explained in section 3.5.

In addition to these 'UML components' diagrams, the CCL plugin adds the following diagram to ArgoUML:

- The *'Atomic Invocation Path Diagram'* is a UML sequence diagram tailored to depict the communication calls invoked by one component. It is explained in section 3.7.

Furthermore, the following standard UML diagram being not part of the 'UML components' approach is discussed here:

- The *'Deployment Diagram'* is optional. It facilitates designing a distributed system as described in section 3.6.

All these diagram are based on the well-known standard UML. Therefore, they are only sketched to some extent. The following sections focus on how to enrich the diagrams of a component model with useful context properties. Each context property is introduced via examples in a certain diagram.

3.2 General Context Properties

Some context properties are only useful in one diagram type, while others can be used in any diagram. This sections lists *general* context properties. Some of them are illustrated in figure 3.1.

'Workflow' : ...reflects the most frequent workflows in which the associated element is used. For example, a requirement in a system could state that *"all components needed by the workflow 'Integrate Two Contracts' must be inaccessible to the 'Web Server' component"*. This requirement can be written down by identifying all of the components involved accordingly. Only the names of the workflows used most often are into account for requirement specification here. 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' : ... describes, in which department(s) or or domain(s) the associated element is used.. It provides an organisational perspective. If necessary, additional context properties of increasing granularity, e.g. 'Role', enable a more detailed specification of legal policies, responsibility, security or access rights requirements. 'Operational Area' illustrates how to model roles or authorization via context properties.

'Personal Data' : It signals whether an element handles data of private nature. Thus, privacy policies, like, *"all components containing 'PersonalData' must not be readable by the web server"* can be specified.

'TransactionStart' : The various component platforms provide relatively rich schemes for the definition of transactional behaviour. In business information systems, pretty much everything is transactional. The specifi-

cation issue therefore boils down to whether new transactions are started by a component or not. Hence, TransactionStart contains the names of the transactions that are started at its element.

'≪Number≫ Amount (in Instances)' : ...describes the estimated number of instances managed by the system. It facilitates establishing reasonable load balance later on as described in [12] and in section 4.3.4. The stereotype ≪Number≫ is used for context properties that describe the 'quality of service' as explained in section 2.1.9.

'≪Number≫ Changes (per Day)' : ...indicates the number of updated, added and removed instances per day. It allows frequently changing data to be distinguished from mostly static data. Again, this information is needed to estimate the network load later on.

Quick Tour $\xrightarrow{goto}$ Chapter 4 If you are not interested in more examples for useful context properties for component-based systems, ride on to chapter 4.

3.3 Enriching Business Type Diagrams With Context Properties

The main purpose of the business type diagram is to create a precise common vocabulary among the people and components involved in the project. For example if 'Contract' means three different things within the business then you need to get this cleared up as early as possible so that everyone is working to the same set of terms with agreed meanings.

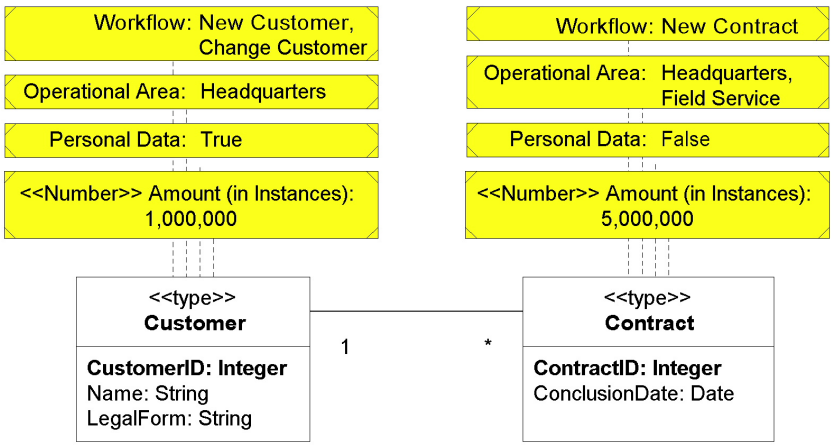

Figure 3.1: Enriching the Business Type Diagram with Context Properties

'Business Type' According to [16], a **business type** represents a business object that is managed by the modelled system. A business type is a concept similar to a class, an abstract data type or a data structure. It describes data that is exchanged between the components of the system.

The system's components exchange business types via interfaces. Each interface can define its own view on a business type. It combines these separate business types into one single, *global* view. A UML class diagram is used to express the business type diagram. Business types, however, are described in less detail than classes. Generally, not all attributes are defined yet, and they lack of methods.

Context Properties The *general* context properties attached to the business types 'Customer',

'Contract', 'Product' and 'ContractArticle' in figure 3.1 are discussed in section 3.2.

3.4 Enriching Component Specification Diagrams with Context Properties

The component specification diagram describes the system's logical component structure, including the interfaces and connectors as discussed in the following sections. As discussed in section 3.1, other modelling techniques for specifying the architecture of a component-based system exist. The context properties suggested in the next sections can be applied to any modelling technique that specify 'components' and 'interfaces'. For instance, the proposed context properties can be applied to UML deployment diagrams (see section 3.6).

3.4.1 Enriching Components With Context Properties

Component specification diagrams

Besides new components, any existing components or other legacy software assets need to be taken into account, as well as any architecture patterns, which will be used. Figure 3.2 shows an excerpt taken from the case study using the component specification diagram of the 'UML components' approach. According to [16], components are specified as classes of the ≪comp spec≫ stereotype (comp spec is an abbreviation of component specification).

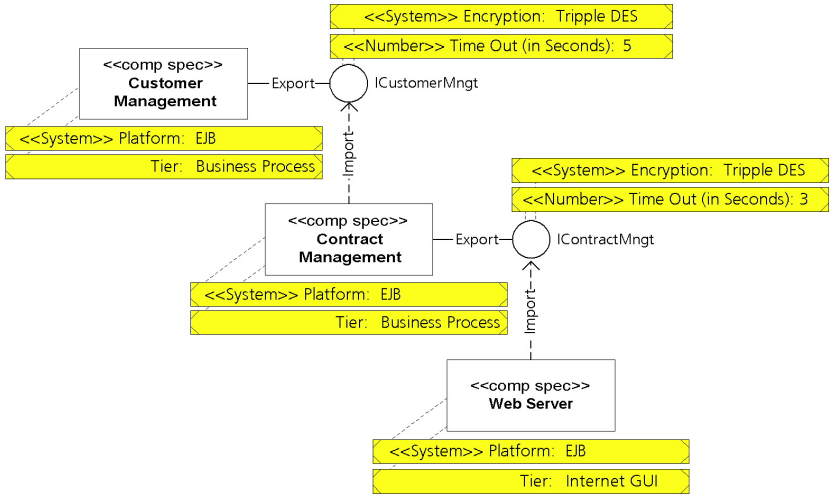

Figure 3.2: Enriching The Component Specification Diagram with Context Properties

Context Properties

Two context properties are suggested for enriching a component diagram. They are demonstrated in figure 3.2:

'Tier' : ... groups all of the components from the same architectural tier. It allows to identify the architecture of a multi-tier system. For instance, 'Client', 'Business' and 'Data' might be useful values of this context property.

'≪System≫ Platform' : ...is helpful in multi-platform systems and expresses which platform each component belongs to. This system property is particularly useful in distinguishing legacy systems from other parts of the system. For example, a component may be located on a mainframe

and, thus, be the subject of a technological restriction, such as asynchronous communication via a message broker.

'≪System≫ TransactionMode' : Current component platforms support four transaction modes as values of this system property: 'Requires a Transaction', 'Requires a New Transaction', 'Supports Transactions' and 'Does not Support Transaction'.

3.4.2 Enriching Interfaces With Context Properties

≪Import≫/≪Export≫ According to [37, 23] a component's structure consists of three parts. The body is encapsulated by two interface types: import interfaces describe which services the component needs to run, while export interfaces contain the services this component offers to other components.

Interfaces In a UML deployment diagram, interfaces are depicted via 'lollipop' stereotypes. They can be refine to ≪Import≫ or ≪Export≫ interfaces as illustrated in figure 3.2.

Context Property: Two context properties are suggested for describing the context of component interfaces:

'≪Number≫ Time Out' : Its values limit the latency between input/output of the corresponding interfaces. Either abstract terms, like 'No', 'Short' 'Long', or precise milliseconds using a discrete model of time can be used. It is a ≪Number≫ context property, since it describes the quality of service.

'≪System≫ Encryption' : Its value(s) describe which encryption mechanism is used. For instance, 'Blowfish'or 'Triple-DES' are well known encryption mechanisms.

For each interface in the component diagram, an interface specification diagram exists as discussed next.

3.5 Enriching Interface Specification Diagrams With Context Properties

Details of an Interface Component specification diagrams (see section 3.4) depict only the names of interfaces. The details of one interface are defined in an 'interface specification diagram'. Each interface specification is a contract with a client of a component object. It defines the details of its contract in terms of the operations it provides, what their signatures are, what effects they have on the parameters of the operations and the state of the component object, and under what conditions these effects are guaranteed. All this information is grouped into a single package for each interface. This is where most of the detailed system behaviour decisions are pinned down. Again, besides the context properties and the context-based constraints most concepts used here are thoroughly described in [16]. An interface specification diagram consists of:

1. A class of the ≪Interface Spec≫ stereotype representing the Interface and defining its operations.
2. If an attribute *a* of an business types *bt* of the business type diagram is used as parameter in one or more operations, this dependency is represented via an ≪info type≫ having the same name as *bt* and showing the attribute *a* used in the operation(s). Details are explained in section 3.5.1.

3. CoCons – the key concept discussed here are context-based constraints. If CoCon is directly associated with an interface it can be specified there. It can be directly associated with the Interface either by naming the interface or via the keyword 'THIS'.
4. The operation's pre- and post-conditions
5. Any additional OCL invariants on the interface information model

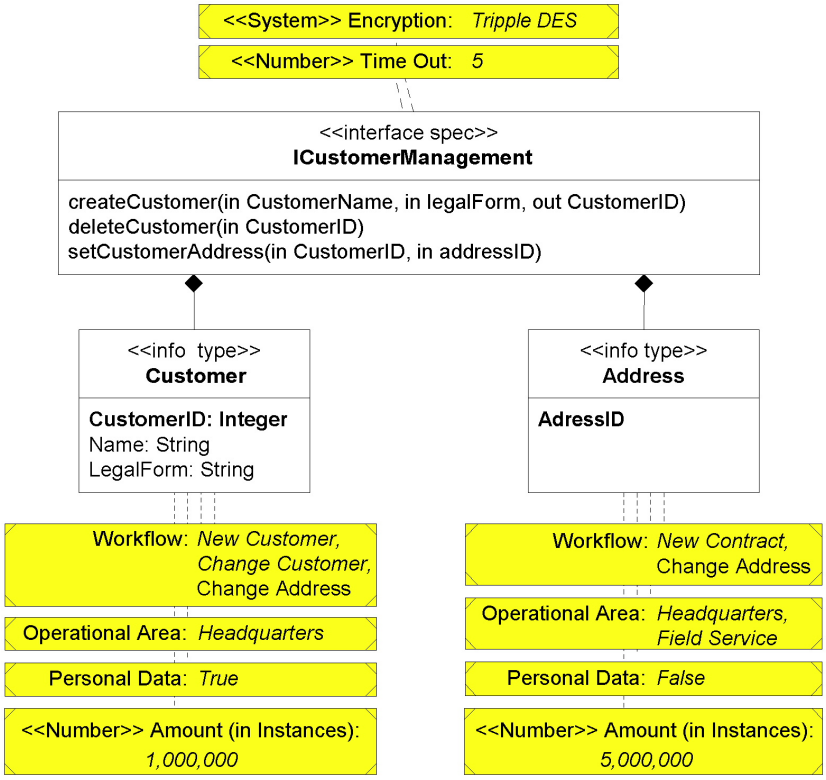

Figure 3.3: Derived Context Property Values Are Printed In Italics

The first two concepts are illustrated in figure 3.3. The following two parts of an interface specification diagram are only sketched here. Again, a detailed description is given in [16].

Operation specification — An operation specification consists of a signature and a pre-condition/post-condition pair.

Signature : — An operation defined on an interface has a signature comprising zero or more typed parameters.

Pre- and post-condition pair : — In UML, operation pre- and post-conditions are defined as constraints on that operation, with the stereotypes «precondition» and «postcondition» respectively.

«Import»/«Export» — In addition to [16], the stereotype «Interface Spec» can be refined as «Export Interface Spec» if it represents an export interface. Likewise, «Import Interface Spec» can be used for import interfaces.

3.5.1 Enriching Interface Information Models with Context Properties

Info Type — Each interface has its own view of the global business types it cares about. Such a view is called 'information type' or **'info type'** and is specified

in the interfaces specification diagram. The info types of an interface specification diagram are called **'interface information model'** [16]. The info types are exclusive to their interface, and represent a projection or refinement of the type information in the business type diagram. Thus, an info type derives the derivable context property values from its business type as explained in section 3.8.

%diss enditemize

3.6 Enriching UML Deployment Diagrams with Context Properties

Modelling Distribution

Arranging the distribution of services and data is a critical task that can ultimately affect the performance, integrity and reliability of the entire system. In most cases, this task is still carried out only in the implementation phase. This approach results in expensive re-engineering of the model after discovering the technological limitations. This frequently leads to adaptations directly on the implementation level alone. Thus, verification of system properties is made more complicated or even impossible due to outdated models. Up to now, distribution decisions are typically taken during implementation. Distribution constraints are introduced in section 4.3. They enable to specify, which component must run on which container in which computer. By writing them down in the model, the system can be checked automatically for distribution problems already during design as described in section 4.3.4.

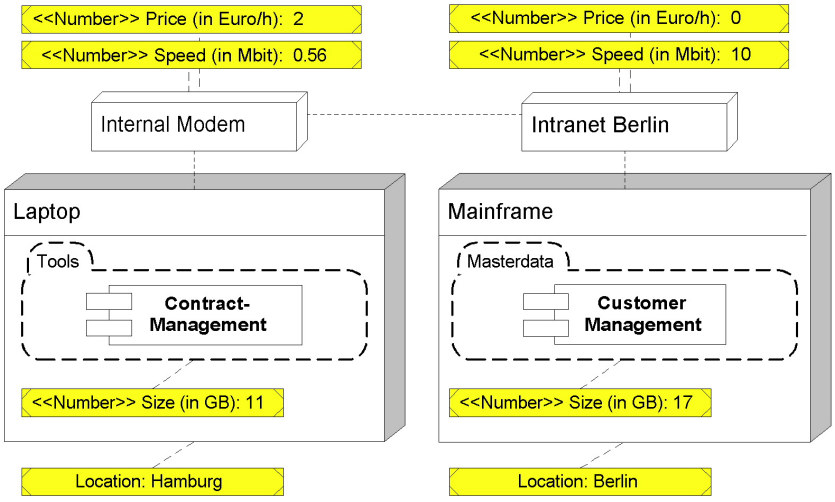

Figure 3.4: The Enhanced UML Deployment Diagram

Enhanced Deployment Diagram

UML deployment diagrams are sometimes turned down – e.g. [26] describes them as 'informal comic'. Applying context-based constraints to them in order to design distributed systems makes them more attractive: Taking distribution decisions demands awareness of hardware and software resources. In [12] we propose to use an enhanced UML deployment diagram for validation of the distribution goals and for architectural reasoning. In addition to the standard UML deployment diagram we suggest to display 'Containers' as packages and - of course - to enrich the diagram with context properties as demonstrated in figure 3.4.

Computers : In UML deployment diagrams, a box symbol denotes a computer.

Containers : A container belongs to one computer and can be depicted via a

UML component of the stereotype ≪Container≫. In figure 3.4, a special component symbol with round corners and a dotted line is used. However, it is not implemented in the CCL plugin for ArgoUML yet.

Connections : A connection links computers and is drawn with the standard UML deployment diagram symbol for connections.

Components : A component is deployed in a container on a computer and is depicted via the standard UML symbol for components.

Types & Instances By working with symbolic names, like 'RAID-Server', the same deployment diagram can be deployed for different customers and cases, and it does not have to be changed if the hardware is replaced. For instance, symbolic names can describe hardware requirements or operating systems. A constraint stated for *one* symbolic computer, e.g. 'Laptop', applies to *all* computers of this kind. By using symbolic names, the network topology model must not be modified if the number of laptops changes. Thus, one computer can describe a whole 'clusters' of similar computers. If distribution constraints differ for computers within one cluster, a symbols of a node instances must represent each computer in this cluster in the deployment diagram.

Context Properties for Computers The following context properties are proposed for computers:

Location : This context property describes, where a computer is installed. In case of mobiles computers, the context property 'Operational Area' might be used.

Client : The context property that describes the software running on a computer does not appear in figure 3.4: 'Client' can be assigned to computers and describes application programs that accesses the component-based system and runs on the computer to which the context property is attached. Client-related information facilitates deciding upon allocation because 'availability' is optimal when all data needed by a client is available on the computer where this application runs on.

Middleware : If heterogeneous component platforms are used, this context property expresses, which product in which version runs on a node.

The hardware is described via ≪Number≫ properties (see section 2.1.9) for establishing reasonable load balance later on. E.g., knowing a connection's 'Speed' can be helpful in taking a distribution decision. The 'Size' specified for each container, like all ≪Number≫ properties, is only an vague estimation.

3.7 The Atomic Invocation Path Diagram

Invocation Path Some of the CoCon types presented section 4 can be only verified during design if the model of a component-based system somehow describes which component invokes which other components. Interfaces contain operations, and components implement interfaces. The invocation of an operation $op()$ of the interface of component c_2 by component c_1 can induce the invocations of other operations. The sequence of invocations induced by calling one operation of one component is called **invocation path** of this operation.

Tailored Sequence Diagram In addition to the 'UML components' diagrams, the CCL plugin adds the **atomic invocation path diagram** to ArgoUML. It is a UML sequence

diagram tailored to depicts how one component c invokes other components if one of c's operations is called. In contrast to normal sequence diagrams, it is restricted to depict the invocation path of only one operation in order to fit on one page or screen. Hence, the consequences of invoking one operation implemented by one component are described in one *atomic* invocation path diagram.

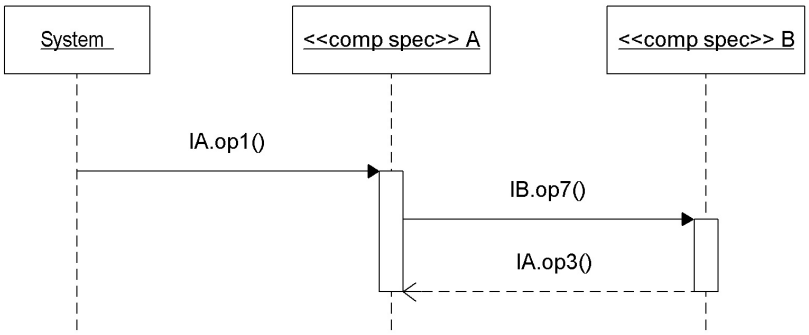

Figure 3.5: The Atomic Invocation Path Diagram

Example For example, figure 3.5 shows the consequences of calling the operation op_1 of the interface IA of the component A. According to figure 3.5, A will always invoke operation op_7 of component B in order to execute its operation op_1.

Other UML Diagrams Of course, other specification techniques can be used to describe invocation paths, too. The 'UML Components' approach uses UML collaboration diagrams called 'component interaction diagram' in order to describe invocation paths. Moreover, the dependency (depicted as dotted arrow) used in component specification diagrams (see figure 3.2) or in deployment diagrams can describe invocation paths. As explained in section 4, several CoCon types need invocation path information in order be verified. Their verification mechanism must be adapted to the techniques used for specifying invocation paths.

3.8 Belongs-To Relations within the UML Components Approach

Quick Tour $\xrightarrow{goto}$ Chapter 4 Quick readers can skip this section and proceed with chapter 4 on page 42. Belongs-To relations are not a key concept of context-based constraints.

Goal: Derive Values Belongs-To relations are introduced in section 2.1.5. If $e_1 \overset{belongs}{\longrightarrow} e_2$ then all values of derivable context properties associated with e_2 automatically apply to e_1. Figure 3.6 proposes belongs-to relations that are useful in the UML components approach.

Metatypes & Elements Figure 3.6 depicts metatypes (also called metaclasses in UML) as rectangles. A belongs-to relation between two meta*types* defines *implicit* belongs-to relation between the instances of these types as explained in section 2.1.5). It does not define implicit belongs-to relations between *all* instances of these metatypes. For instance, not *all* components belong to *all* computers. Instead, only those elements of the metatype 'component' belong to an elements of the metatype 'computer' that fulfil the

belongs-to criteria. Each belongs-to relation is described here, and for each belongs-to relation the belongs-to criteria is written in *italics*:

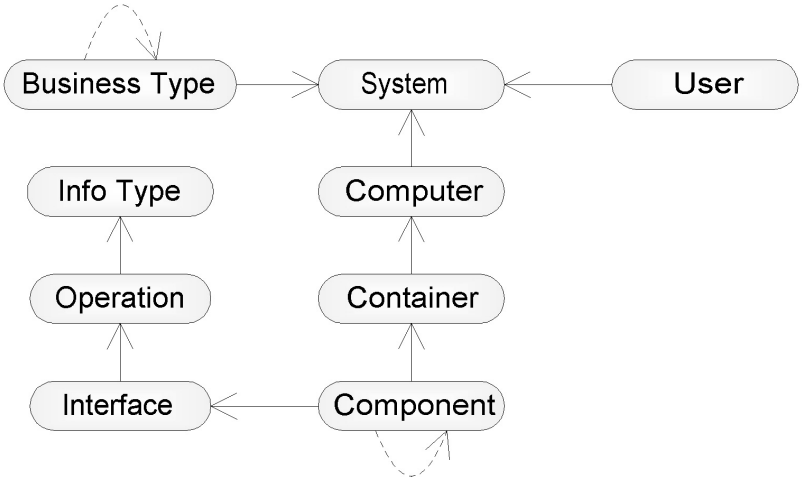

Figure 3.6: Proposed Hierarchy of Belongs-To Relations between Meta-classes of the UML Components Approach

Components $\overset{belongs}{\longrightarrow}$ *Container :* A component *installed in* a container belongs to this container.

Container $\overset{belongs}{\longrightarrow}$ *Computer :* A container *installed in* a computer belongs to this computer.

Computer $\overset{belongs}{\longrightarrow}$ *System :* A computer belongs to *its* system.

User $\overset{belongs}{\longrightarrow}$ *System :* A user *using the system* belongs to this system.

Business Type $\overset{belongs}{\longrightarrow}$ *System :* A business type belongs to *its* system.

Info Type can belong to a Business Type : An info type belongs to the business type *on which it is a view and if the view shares the same context as its business type.* If the info type is identical to the business type then the context property values of the info type can be derived from the business type. However, the view can ignore some of the attributes of the business type or handle only a part of the business type instances. Hence, the view may not reflect all contexts of the viewed business type. If the info type is not identical to the business type then it can only *manually* be defined whether one info type derives all derivable context property values from the business type on which it is a view. A belongs-to relation from info type to business type can only be defined if the belongs-to criteria (in *italics*) can be automatically decided. Thus, no belongs-to relation is defined on metalevel – no arrow points from info type to business type in figure 3.6.

Operation $\overset{belongs}{\longrightarrow}$ *Info Type :* A operation belongs to the info type(s) *if at least one attribute of the info type is used as parameter of the operation.*

Interface $\overset{belongs}{\longrightarrow}$ *Operation :* An interface belongs to *its* operation(s).

Component $\overset{belongs}{\longrightarrow}$ *Interface :* A component *that implements an interface* belongs to this interface.

'implicit' A CoCon-capable modelling tool must be able to automatically identify

an 'implicit' belongs-to relation. For example, it must be able to automatically evaluate the belongs-to criteria.

Composition — Next, extra belongs-to relations are described that are not part of the UML components approach. They are depicted as bended and dotted arrows in figure 3.6, and they all describe **composition** of elements. The UML components approach as described in [16] does not handle composed components or composed business types. Hence, the composition described next is an additional concept to the UML components approach. It is reflected here because other component specification techniques, like [23], handle composition.

Business Type $\overset{belongs}{\longrightarrow}$ *Business Type :* A business type bt_{part} implicitly belongs to another business type $bt_{composite}$ if bt_{part} is a part of $bt_{composite}$ due to an aggregation between bt_{parent} and $bt_{composite}$.

Component $\overset{belongs}{\longrightarrow}$ *Component :* A component c_{part} implicitly belongs to another component $c_{composite}$ if c_{part} is a part of $c_{composite}$ due to an aggregation between c_{parent} and $c_{composite}$.

Skipping Unused Metatypes — Any element (not only computers, business types or users) belongs to the system. However, belongs-to relations are not defined between each element and the system. Instead, the context property values should be derived along the belongs-to hierarchy illustrated in figure 3.6 in order to facilitate handling the context property values. For example, a component belongs to a system. If the system model tells, which component resides in which container on which computer, than a component belongs to the system due to *Components* $\overset{belongs}{\longrightarrow}$ *Container* $\overset{belongs}{\longrightarrow}$ *Computer* $\overset{belongs}{\longrightarrow}$ *System*. A belongs-to relation *Component* $\overset{belongs}{\longrightarrow}$ *System* is only needed if containers and computers are not specified in the model. If the model misses any of the metatypes in the belongs-to hierarchy in figure 3.6 then the missing metatypes can be skipped. For example, *Component* $\overset{belongs}{\longrightarrow}$ *System* can be used instead of *Components* $\overset{belongs}{\longrightarrow}$ *Container* $\overset{belongs}{\longrightarrow}$ *Computer* $\overset{belongs}{\longrightarrow}$ *System* if neither containers nor computers are specified in the system model.

Interface ≡ Interface — The same metatype can be depicted in different diagrams. The UML components approach even uses different UML metaclasses for the same metatype 'interface' in different diagrams : in the 'component specification diagram' an interface is depicted via a 'lollipop' that represents an instance of the UML metaclass 'interface', while the 'interface specification diagram' uses the UML metaclass 'class' of the stereotype ≪interface spec≫ to depict the same interface. This can result in inconsitency because the same interface is represented via several, redundant model elements. Hence, a modelling tool should be able to assign the same context property values to the same interface, regardless whether it is depicted via an lollipop or via a class. This is not a belongs-to relation because even values of non-derivable context properties must be copied from one model element to the other one representing the same interface. Modelling tools must be able to handle this pitfall in the UML components approach.

Types or Instances — The metatypes in figure 3.6 either all represent types or all represent instances. Both types and instances can exist in the model of a component-based systems. However, types do not belong-to their instances, and instances do not belong-to their types. For example, a component *type* belongs-to the container *type* in which it is deployed according to figure

3.6. But a component *instance* does not belong-to the container *type* in which its type is deployed. As explained in section 2.1.4, the context property values associated with a type define the range of values that can be associated with the instances of this type: context property values assigned to an instance must be contained in the context property values assigned to the type of this instance according to the type-instance constraint on context property values.

Hey - Shouldn't That Arrow Point In The Other Direction?

All belongs-to relations in figure 3.6 are directed. Nevertheless, why don't they point in the other direction? For example, why does a component belong to the interface it implements, and why does an interface not belong to the component that implements it? The directions of belongs-to relations proposed here have been chosen in order to achieve an elegant belongs to hierarchy. Any belongs-to relation can be specified the other way 'round, too. Nevertheless, a belongs-to hierarchy should avoid bi-directional belongs-to relations because context property values will be propagated in both directions then. As all elements somehow belong to each other, one value associated with one element finally will apply to all elements - all elements will have the same context property values. This impedes the concept of context-based constraint because each context condition will select either all elements or none if they all have the same context property values. Hence, all belongs-to relations of one belongs-to relation type should be directed in the same direction.

To Which Element Should I Associate The Value?

The reason for defining a belongs-to relation is not to just to show that somehow one element belongs to the other one, but to derive the context property values from one element to the other one in a way that facilitates maintaining the values. If one element changes its context property values then this change is automatically propagated to the other elements that belong to this element. The belongs-to hierarchy in figure 3.6 enables to derive one value along *several* transitive belongs-to relations. This enables the designer to associate a context property value with the element that is as high as possible in the belongs-to hierarchy. It must be *associated* only once and, thus, is *derived to* probably many elements. Hence, redundant and possibly inconsistent context property values can be avoided, and the comprehensibility is increased.

Avoiding Bad Design

Inter-value constraints (see section 2.1.3) must be defined for a context property whose values can contradict each other. An example is explained in section 4.5.6, where an inter-value constraint forbids 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. According to the belongs-to hierarchy suggested here, *all* components deployed in a container must not handle personal data if the value 'False' of the context property 'Personal Data' is associated with the container and if 'Personal Data' is defined as a derivable context property. The inter-value constraint defined in section 4.5.6 is violated if one component in this container does handle personal data. 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? The best answer is: in that case, no value of 'Personal Data' should be associated with the container at all. If some components in the container handle personal data and some 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, 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.

Example for Belongs-To Relations

All context property values printed in italics in figure 3.3 are derived from other diagrams via belongs-to relations: the values of '≪System≫ Encryption' and '≪Number≫ Timeout (in Seconds)' are derived from figure 3.2, while the values of 'Operational Area', 'Personal Data' and '≪Number≫ Amount (in Instances)' are derived from figure 3.1. But, the values of 'Workflow' differ for the '≪info type≫ Customer': the values '*New Customer*' and '*Change Customer*' are printed in italics because they are derived from the '≪type≫ Customer' in figure 3.1 via the belongs-to relation **Info Type** $\overset{belongs}{\longrightarrow}$ **Business Type** explained in section 3.8. Additionally, a new value is associated with '≪info type≫ Customer' that is not derived via a belongs-to association: 'Change Address' (not in italics) is newly associated with the '≪info type≫ Customer' in this interface specification diagram.

I Still Don't Like It

Belongs-To relations are not a key concept of context-based constraints. Instead, the only improve the handling of context property values. Don't use belongs-to relations if you see no benefit in deriving the values of one element from another element to which it belongs.

4. The Context-Based Constraint Language CCL

The notion of context-based constraints (CoCons) is explained in section 2.2. The **C***ontext-based* **C***onstraint* **L***anguage CCL* for components consists of different CoCon types as explained in the next sections. Future research will result in additional CoCon types.

CoCon Family — A **CoCon family** groups CoCon types with related semantics. For instance, `ACCESSIBLE TO` CoCons, `WRITEABLE BY` CoCons and `READABLE BY` CoCons belong to the family of Accessibility CoCons. Conflicting requirements can only be detected automatically for CoCons of the same family. In each of the oncoming sections, one CoCon family is introduced, and its automatically detectable conflicts are discussed.

Element Types — A system or a system model contains different element types. This paper focuses on applying CCL to the model element types used in the 'UML components' approach. Applying CCL to other element types than those listed in chapter 3 is topic of future research.

Focus: Components — Most examples given in the next sections only refer to one element type: 'components' are the key concept in the continuous software engineering paradigm (CSE, see section 1.1). Hence, this paper concentrates on applying CCL to components in order to integrate CCL into CSE.

`(NOT | ONLY)` — Each CoCon type can combined with the **CoCon-type-condition** 'NOT' or 'ONLY' after the keyword MUST. For example, the CoCon type `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-type-conditions of one CoCon type in the next sections.

4.1 Accessibility CoCons

4.1.1 The Notion of Accessibility CoCons

Goal: Security — CoCon types of the accessibility family enforce read and write permissions. They determine if and how elements can access other elements. Thus, they facilitate handling security requirements.

Constrained Element Types — The target set of an accessibility CoCon can contain any element type that can access other elements, such as 'components'. As well, the scope set of accessibility 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 accessibility CoCons are discussed here. The application-specific security mechanism used to enforce the CoCon can be specified in the CoCon's `ACTION` attribute.

4.1.2 Accessibility CoCon Types

`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. The Enterprise Java Beans standard only distinguishes

between accessible and inaccessible components. Other middleware platforms, like CORBA, allow for more subtle distinction:

`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.

For example, an accessibility CoCon can state that `ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes' MUST NOT BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'`.

Adressing Indirect Invocation — An accessibility CoCon can state that the component c_{Start} is not allowed to acess the component c_{End}. This statement seems to allow that component c_{Start} can invoke c_{17}. But, if c_{17} invokes c_{End} after c_{Start} has invoked c_{17} then c_{Start} has *indirectly invoked* c_{End}. An accessibility CoCon must define if indirect invocation is allowed or not. If indirect invocation is not allowed then different strategies exist for preventing it. A chain of components that indirectly invoke eachother is called *invocation path* here. In order to dicsuss blocking strategies, the components on an invocation path containing at least three components are listed now:

- An invocation path from the component c_{Start} to the component c_{End} exists
- On this invocation path, one or more components $c_i, ..., c_k$ exist(s). c_{Start} invokes c_i, and c_k invokes c_{End}. If only three components are contained in this invocation path then $c_i = c_k$.

Blocking Strategies — If an accessibility CoCon forbids c_{Start} to indirectly invoke c_{End} then the invocation path can be blocked at the following positions:

Optimistic Blocking : c_k is not allowed to invoke c_{End}, but all other components are allowed to invoke each other.

Pessimistic Blocking : c_{Start} is not allowed to invoke c_i.

Manually Defined Blocking : The system administrator defines at which component of $c_i, ..., c_k$ the invocation is blocked.

The blocking strategie must be defined via the `ACTION` attribut. Besides 'Enable Indirect Invocation', the values 'Optimistic Blocking', 'Pessimistic Blocking' can be enforced automatically.

4.1.3 Detectable Conflicts of Accessibility CoCons

Detectable Violations — As explained in section 2.2.2, one CoCon can contradict other CoCons. The elements e_i, e_k, e_m are target or scope set elements of accessibilty CoCons with $i \neq k \neq j$. Conflicting accessibility CoCons can be automatically detected if one of the following rules is violated:

The Alias 'X-ABLE' The rules listed above are generic because of the alias `X-ABLE`. It must be replaced either with one of the four accessibility CoCon types `ACCESSIBLE TO`, `READABLE BY`, `WRITEABLE BY`or with `EXECUTEABLE BY`. If the alias `X-ABLE` appears more than once in one generic rule then it must be replaced with the same accessibility CoCon type each times.

1. No element e_i can be both `X-ABLE` and `NOT X-ABLE` by any e_k
2. No element e_i can be `X-ABLE` by e_m if it is `ONLY X-ABLE` by e_k but not `ONLY X-ABLE` by e_m
3. No element e_i can be `ONLY X-ABLE` by e_m if it is `ONLY X-ABLE` by e_k and $e_i^{ScopeSet}$ is not element of the same scope set as e_k
4. No element e_i can be `NOT X-ABLE` by itself - see the example in section 4.1.5.
5. No element e_i can be both `ACCESSIBLE TO` and `NOT READABLE BY` any e_k
6. No element e_i can be both `ACCESSIBLE TO` and `NOT WRITEABLE BY` any e_k
7. No element e_i can be both `ACCESSIBLE TO` and `NOT EXECUTABLE BY` any e_k
8. No element e_i can be both `ACCESSIBLE TO` and `NOT REMOVEABLE BY` any e_k
9. No element e_i can be both `NOT ACCESSIBLE TO` and `READABLE BY` any e_k
10. No element e_i can be both `NOT ACCESSIBLE TO` and `WRITEABLE BY` any e_k
11. No element e_i can be both `NOT ACCESSIBLE TO` and `EXECUTABLE BY` any e_k
12. No element e_i can be both `NOT ACCESSIBLE TO` and `REMOVEABLE BY` any e_k

4.1.4 Verifying Accessibility CoCons

Specifying Communication Calls Many specification techniques exist for describing how components invoke each other. If communication calls are specified in the model via techniques for expressing communication calls then this model can be automatically checked for compliance with accessibility CoCons. However, the algorithms for checking the model depends on the modelling technique used for specifying communication calls. Section 5.2 discusses four modelling techniques for depicting communication calls in UML: sequence diagrams, collaboration diagrams, deployment diagrams or class diagrams can be checked for compliance with accessibility CoCons during design.

4.1.5 Examples for using Accessibility CoCons

Privacy Policy In this section, an `ACCESSIBLE TO` CoCon illustrates the case where the target set and the scope set of a CoCon can overlap. Moreover, the example shows how to detect whether a system complies with an accessibility CoCon. A requirement might state that *"no component belonging to the operational area 'Controlling' is allowed to access components that*

handle personal data". This statement can be specified in CCL in the following way:

```
ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes'
MUST NOT BE ACCESSIBLE TO
ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'
WITH ACTION = 'Pessimistic Blocking'
```

Self-Access If a component has the value 'Yes' for its context property 'Personal Data' *and* the value 'Controlling' for its context property 'Operational Area' then it belongs *both* to the target set and to the scope set of the CoCon. This is absurd, of course. It means that this component cannot access itself. Such bad design is detected via the rule number four in section 4.1.3. Every component involved in this conflict 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 they cannot comply with this requirement.

Blocking Strategy If the component C belongs to the controlling department and the component P handles personal data, than this CoCon demands that C does not invoke P. Moreover, the CoCon demands that C does not indirectly invoke P via other components, like A. For example, an *invocation path* between C and P exists if C invokes A and A invokes P. The value 'Pessimistic Blocking' of the `ACTION` attribute defines that every invocation call from C and A must be blocked as explained in section 4.1.2. In case of the `ACTION` 'Optimistic Blocking', C is allowed to invoke A, but A is not allowed to invoke P if this invocation of P was initiated by C invoking A. In order to enforce optimistic blocking, it must be defined which component initiates an invocation path. As explained next, optimistic blocking cannot be *transparently* integrated into Enterprise Java Beans.

EJB According to [6, 59], not all accessibility CoCons can transparently be monitored in EJB systems at runtime because an EJB component cannot identify the caller of its interface. Therefore, the context property values of the caller cannot be determined without modifying the source code of the components. Moreover, the component that initiated the invocation path cannot be identified. The current user, however, can be identified at runtime. Therefore, the scope set of an accessibility CoCon should refer to users instead of components if the CoCon shall be applied to EJB systems at runtime. The example given above can be stated in the following EJB-compatible way:

```
ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes'
MUST NOT BE ACCESSIBLE TO
ALL USERS WHERE 'Role' CONTAINS 'Controller'
```

4.2 Communication CoCons

Communication CoCons are are typically monitored at runtime or during configuration because they describe how to handle communication calls between components.

4.2.1 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 an communication CoCon then a **context-aware service** is invoked. Most of the context-aware services suggested

here handle non-functional requirements. They mostly don't modify the call's content. Instead, they add additional features, like logging the call or encrypting it.

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.

4.2.2 The Communication CoCon Types

The family of communication CoCons consists of several CoCon types:

`ENCRYPTED WHEN CALLING`

An **ENCRYPTED WHEN CALLING** CoCon specifies that a context-aware encryption service encrypts each communication call between a component in its target set and a component in its scope set. The attribute `ACTION` determines the encryption mechanism.

`ERRORHANDLED WHEN CALLING`

An **ERRORHANDLED WHEN CALLING** CoCon specifies that a context-aware error handling service catches the exceptions that are thrown when a component in its target set invokes a component in its scope set. The attribute `ACTION` determines the error handling mechanism.

`HANDLED WHEN CALLING`

Besides the context-aware services described above, other services may be useful. In order to be able to handle them, a generic communication CoCon type is defined. A **HANDLED WHEN CALLING** CoCon is generic because its impact is only defined in its attribute `ACTION`. The `ACTION` attribute describes or context-aware service that is executed each time when a component in its target set invokes a component in its scope set. This `ACTION` statement can be realised a connector, a mediator or a wrapper at runtime.

`LOGGED WHEN CALLING`

A **LOGGED WHEN CALLING** CoCon specifies that a context-aware logging service logs each communication of a component in its target set with a component in its scope set. The attribute `ACTION` determines the logging mechanism used.

`REDIRECTED WHEN CALLING`

A **REDIRECTED WHEN CALLING** CoCon specifies that a context-aware redirection service: each time when a component in its target set invokes a component in its scope set the communication call is redirected to the destination specified in its `ACTION` attribute.

The following communication CoCon types are typically applied during configuration:

`PROTECTED BY A TRANSACTION WHEN CALLING`

A **PROTECTED BY A TRANSACTION WHEN CALLING** CoCon specifies that the communication calls from a component in its target set to a component in its scope set must be protected by a transaction mechanism. The attribute `ACTION` determines the transaction mechanism used.

`SYNCHRONOUSLY CALLING`

A **SYNCHRONOUSLY CALLING** CoCon specifies that the elements of the target set must synchronously invoke the elements in the scope set. The attribute `ACTION` determines the communication mechanism used, like RPC.

`ASYNCHRONOUSLY CALLING`

An **ASYNCHRONOUSLY CALLING** CoCon specifies that the elements of the target set must asynchronously invoke the elements in the scope set. The attribute `ACTION` determines the communication mechanism used, like message brokers.

4.2.3 Detectable Conflicts of Communication CoCons

As explained in section 2.2.2, one CoCon can contradict other CoCons. Conflicting communication CoCons can be automatically detected if one of the following rules is violated.

`The Alias 'X-ED WHEN CALLING'`

The elements e_i, e_k, e_m are target or scope set elements of communication CoCons with $i \neq k \neq j$. The rules listed here are generic because of the alias `X-ED WHEN CALLING`. It must be replaced with one of the communication CoCon types. If the alias `X-ED WHEN CALLING` appears more than once in one generic rule then it must be replaced with the same communication CoCon type each time.

1. No element e_i can be both `X-ED WHEN CALLING` and `NOT X-ED WHEN CALLING` any e_k
2. No element e_i can be `X-ED WHEN CALLING` e_m if it is `ONLY X-ED WHEN CALLING` e_k, but not `ONLY X-ED WHEN CALLING` e_m
3. No element e_i can be `ONLY X-ED WHEN CALLING` e_m if it is `ONLY X-ED WHEN CALLING` e_k and e_m is not element of the same scope set as e_k
4. No element e_i can be both `SYNCHRONOUSLY CALLING` and `ASYNCHRONOUSLY CALLING` any e_k

4.2.4 Verifying Communication CoCons

Checking... Most of the communication CoCons are best applied during configuration or at runtime.

...During Design The 'UML Components' approach for modelling component-based systems does not specify encryption, error handling, logging, wrapping, redirection, caching or transactions. These concepts can only be verified in the model if they are specified in the model. Any modelling approach that specifies these concepts can be checked for whether the model violates communication CoCons.

...During (Re-)Configuration If the configuration files of the middleware platform control encryption, error handling, logging, wrapping, redirection, caching or transactions then communication CoCons can be verified during configuration. For instance, current component platforms determine the transaction mode of each component in configuration files as 'Requires a Transaction', 'Requires a New Transaction', 'Supports Transactions' and 'Does not Support Transactions'. These transaction modes can be specified in the `ACTION` attribute of a `PROTECTED BY A TRANSACTION WHEN CALLING` CoCon. However, a CoCon cannot be checked during configuration if its context condition refers to a context property whose value changes at runtime.

...At Runtime Dynamic context property values can only be checked at runtime. A framework for intercepting communication calls in order to monitor them for compliance with CoCons is described in [6, 13, 35, 59].

4.2.5 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' WITH ACTION = 'Log To File C:\\IntegrateTwoContracts.log'`. If a communication call belongs to the workflow 'Integrate Two Contracts' then a context-aware logging-service is invoked which writes a log-entry in the log file at runtime.

4.3 Distribution CoCons

4.3.1 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 [12] – some of their names have been changed meanwhile.

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' vs. 'Load Balance'

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. Different criteria for identifying which element relates to which other element exist. Typically, objects that interact heavily are located together. However, it can be important to locate objects together even if they do not interact at all. Availability can be improved by grouping related objects into *subject-specific* clusters and allocating or replicating the whole cluster to the computer(s) where it must be available.

Subject-Specific Clusters

Objects that interact heavily must be located together. Existing configuration languages, like D^2AL ([4] or 'δarwin' [48] group *collaborating* objects that are directly linked via associations on the design level or directly invoke each other at runtime. However, there may be 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 collaborate or not. Distribution Cocons allocate objects together because of shared context instead of direct collaboration. Distribution Cocons assist in grouping related objects into **subject-specific clusters** and define how to replicate or cache the whole cluster to client sites in order to establish a reasonable trade off between 'availability' and 'load balance'. Other approaches for handling subject-specific clusters exist. For instance, 'dynamic grouping' proposed by [64] defines an 'organizational concept space' consisting of a *similarity network* and an *interest matrix*. The similarity and interest values are extracted by learning from user behaviour. In addition to automatically extracted knowledge about how to cluster objects, distribution CoCons can express intellectually defined availability requirements.

Recording The Design Rational

After distributed applications became popular and sophisticated in the 80s, over 100 *programming languages* specifically for *implementing* distributed applications were invented according to [3]. 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 systems 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.

4.3.2 Distribution CoCon Types

`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, but barely considered in middleware platforms nowadays. As explained in [29], replication can be realised with current middleware platforms. The following CoCon types 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 their allocation – their allocation is specified via `ALLOCATED TO` CoCons – to the elements in its scope set via the replication mechanism given in the CoCon's `ACTION` attribute.

`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 via the replication mechanism given in the CoCon's `ACTION` attribute.

4.3.3 Detectable Conflicts of Distribution CoCons

As explained in section 2.2.2, one CoCon can contradict other CoCons. Conflicting requirements can be automatically detected if one of the following rules is violated. The elements e_i, e_k, e_m are target or scope set elements of distribution CoCons with $i \neq k \neq j$.

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

7. No element e_i may be both `NOT SYNCHRONOUSLY REPLICATED TO` e_k and `SYNCHRONOUSLY REPLICATED TO` e_k.
8. No element e_i may be both `NOT ASYNCHRONOUSLY REPLICATED TO` e_k and `ASYNCHRONOUSLY REPLICATED TO` e_k.
9. No element e_i may be both `SYNCHRONOUSLY REPLICATED TO` e_k and `ASYNCHRONOUSLY REPLICATED TO` e_k.
10. No element e_i may be `ALLOCATED TO` $e_i^{ScopeSet}$ if it is `ONLY ALLOCATED TO` $_k$, but not `ONLY ALLOCATED TO` e_m
11. No element e_i may be `ONLY ALLOCATED TO` e_m if it is `ONLY ALLOCATED TO` e_k and e_m is not element of the same scope set as e_k

4.3.4 Verifying Distribution CoCons

Specifying Distribution... Many specification techniques exist for describing a distributed component-based system. As discussed in section 3.6, a distribution model for component-based systems should identify components, nodes, connections and containers:

A node : is a computer.

A container : belongs to one node. It can contain components. Considering containers in distribution design is optional - if the specification technique ignores containers then components are allocated to nodes directly.

A connection : links nodes.

...via UML Deployment Diagrams These concepts can be specified in UML deployment diagrams as explained in section 3.6. If needed, containers can be depicted in UML deployment diagrams via stereotyped components as illustrated in figure 3.4.

Verifying Availability This section explains that availability can be *verified*, while load balance only can be *estimated*. A component is available at a container or a computer if it is deployed there. In order to verify whether the deployment diagram complies with the distribution CoCons given, each component involved in a distribution CoCon must be checked if it is correctly allocated as defined by its distribution CoCon. A system complies with distribution Cocons if none of its components violates the distribution CoCons and if no conflict between the distribution CoCons is detected as described in section 4.3.4.

Estimating Load Balance Additionally, the system's load balance can be estimated as discussed next – this is rather an 'architectural reasoning' than a 'verification' topic. The numbers representing the system load in figure 3.4 via ≪Number≫ Context Properties (see section 2.1.9) are either a result of estimations and calculations as proposed here, of a simulation as suggested by [40], or of (maybe prototypical) runtime metrics. Approximation based on common sense and experience is the quickest and most vague way to figure out the values of ≪Number≫ Context Properties. They can be investigated by asking domain experts questions like *"how often is each workflow executed?"* during analysis. Then constraints, like *"The 'Size' of a container must be greater or equal to the sum of the '≪Number≫ Amount' of all component instances in the containers of this node"*, can be defined in order to calculate the load approximation. In this example, the number of instances of each component is taken from the '≪Number≫ Amount'

property in figure 3.2. In depth load approximation via ≪Number≫ context properties is a topic of future research, though.

Trade of between Load Balance & Availability

Combining CCL and UML deployment diagrams assists in detecting distribution problems before implementation. The overview perspective assists in identifying where to change the model until a satisfactory trade off between the contradicting goals of distribution design is achieved. Furthermore, combining CCL and UML deployment diagrams serves in discussing achievement of distribution requirement and in finding a reasonable trade off between 'availability' and 'load balance'. For instance, by knowing which component is needed by which workflow it can be discussed within the developer team or with the customers, which distributed transaction is facilitated or impeded by the current distribution design. Only 'Availability' can be automatically verified by checking distribution CoCons. In addition, 'load balance' can be *estimated*, not verified. By expressing both distribution goals in the model, a trade of between load balance and availability can be found.

In addition to model checking, a component-bases system can be checked for compliance with distribution CoCons during (re-)configuration and at runtime.

During (Re-)Configuration

The distribution of a component-based system is stored in configuration files. The formats of configuration files differ for each middleware platform. Thus, a parser must be implemented for each middleware platform that extracts the distribution model from the configuration files in order the check the compliance of the system with the distribution CoCons given.

At Runtime

Typically, the distribution of components does not change at runtime. It is only necessary to check compliance of the system with distribution at runtime, if the components involved change their allocation at runtime.

4.3.5 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'`

4.4 Information-Need CoCons

4.4.1 The Notion of Information-Need CoCons

Goal: Information Logistics

The central problem of information supply today is no longer quantity but quality: the right information has to be filtered out of a flood of data. The individuals are only interested in a tiny fraction of the available data. The aim of **information logistics** ([36]) is to provide the *right* information *when* it is required and *where* it is required. In order to achieve this goal, the *information need* must be defined. This section suggests to define information need via CoCons.

Information Need CoCons

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.

Related Research

According to [53], 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 ([32]) or recommender systems ([49]). Many other approaches exist to extract users' 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. Information-Need CoCons cannot replace these approaches. Instead, a domain expert can intellectually define additional information need via CoCons.

4.4.2 The Information-Need CoCon Types

The family of information-need CoCons consists of the following CoCon types:

`INFORMED OF` CoCons

A `(NOT | ONLY)` **INFORMED OF** CoCon specifies that the elements in its target set are `(NOT | ONLY)` interested in the scope set elements – the target set elements must `(NOT | ONLY)` be informed of the scope set elements. The CoCon attribute `ACTION` describes when and how to inform the target set elements.

What Information?

The aim of **information logistics** ([36]) 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.

When?

The question *when* the information is required can be answered in three ways. First, the CoCon's `ACTION` attribute can describe when and how to deliver the information, for instance, it can demand to deliver the information 'immediately via SMS'. Second, the target set specification describes in which context the information is needed. For example, users `WHERE 'Local Time' = '08:15'` can be specified to be interested. Thus, the target set elements can be selected via a context condition that describes *when* information need exists. Third, the time of interest can depend on the interesting element. Hence, a scope set context condition can refer to the time context of an interesting element, like `DOCUMENTS WHERE 'Publication Year' = '2002'`.

Where?

Likewise, the question *where* the information is required can be specified in a target set context condition and in a scope set context condition. In the target set, a context condition can describe *where* the interested element requires information. In the scope set, a context condition can describe *where* the interesting elements reside. In addition, the attribute `ACTION` can specify where the information must be delivered to.

User-Independent Information Need

An `INFORMED OF` CoCon describes *who* is interested in *what*. Its target set contains the interested elements (*who*). Typically, it contains users. Nevertheless, if it doesn't matter *who* is interested then it can be ignored when expressing information need as discussed next.

`AS INTERESTING AS` CoCons...

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

vice versa. An `AS INTERESTING AS` CoCon does not define *who* is interested in its target or scope set elements. Instead, this can be defined via `INFORMED OF` CoCons. Nevertheless, the CoCon attribute `ACTION` of an `AS INTERESTED AS` CoCon describes when and how to inform whomever is interested.

...Can Be Expanded

An `AS INTERESTING AS` CoCon can be changed into an `INFORMED OF` CoCon by adding the definition of who is interested. In the following table, the alias $ELEMENTS$ can be replaced by any target or scope set specification.

Ignoring Who Is Interested	Considering Who Is Interested
$ELEMENTS_A$ `MUST (NOT | ONLY) BE AS INTERESTING AS` $ELEMENTS_B$	$ELEMENTS_C$ `MUST (NOT | ONLY) BE INFORMED OF` $ELEMENTS_A$ `OR` $ELEMENTS_B$

`AVAILABLE TO ANYONE INTERESTED IN` CoCons

A `(NOT | ONLY)` **`AVAILABLE TO ANYONE INTERESTED IN`** CoCon specifies that any element in its target set must be `(NOT | ONLY)` be available to anyone interested in any scope set element. In contrast to `AS INTERESTING AS` CoCons, the `AVAILABLE TO ANYONE INTERESTED IN` CoCons are unidirectional – whoever is interested in its scope set element is also `| (NOT | ONLY)` interested in all its target set elements, but not vice versa. Again, the CoCon attribute `ACTION` describes when and how to inform whomever is interested.

Constrained Element Types

The target set of an `INFORMED 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. Usually, the scope set contains 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.

4.4.3 Detectable Conflicts of Information-Need CoCons

As explained in section 2.2.2, one CoCon can contradict other CoCons. Conflicting information-need CoCons can be automatically detected if one of the following rules is violated:

The Alias '`AS IMPORTANT AS`'

The rules listed here are generic because of the alias `AS IMPORTANT AS`. It must be replaced either with `AS INTERESTING AS` or with `AVAILABLE TO ANYONE INTERESTED IN` . The elements e_i, e_k, e_m are target or scope set elements of information-need CoCons with $i \neq k \neq j$.

1. No element e_i can be both `INFORMED OF` and `NOT INFORMED OF` any e_k.
2. No element e_i can be `INFORMED OF` e_k if it is `ONLY INFORMED OF` e_m, but not `ONLY INFORMED OF` e_k.
3. No element e_i can be `ONLY INFORMED OF` e_k if it is `ONLY INFORMED OF` e_m and e_k is not element of the same scope set as e_m.
4. No element e_i can be `NOT INFORMED OF` itself.
5. No element e_i can be both `AS IMPORTANT AS` and `NOT AS IMPORTANT AS` any e_k.

6. No element e_i can be `AS IMPORTANT AS` e_k if it is `ONLY AS IMPORTANT AS` e_m, but not `ONLY AS IMPORTANT AS` e_k.
7. No element e_i can be `ONLY AS IMPORTANT AS` e_k if it is `ONLY AS IMPORTANT AS` e_m and e_k is not element of the same scope set as e_m.
8. No element e_i can be `NOT AS IMPORTANT AS` itself.
9. If the element e_i must be `INFORMED OF` e_k and e_m is `AS IMPORTANT AS` e_k then e_i must be `INFORMED OF` e_m, too.
10. If the element e_i must be `INFORMED OF` e_k and e_k is `AS INTERESTING AS` e_m then e_i must be `INFORMED OF` e_m, too.
11. If the element e_i must be `NOT INFORMED OF` e_k and e_m is `AS IMPORTANT AS` e_k then e_i must be `NOT INFORMED OF` e_m, too.
12. If the element e_i must be `NOT INFORMED OF` e_k and e_k is `AS INTERESTING AS` e_m then e_i must be `NOT INFORMED OF` e_m, too.
13. If the element e_i must be `ONLY INFORMED OF` e_k and e_m is `AS IMPORTANT AS` e_k then e_i must be `ONLY INFORMED OF` e_m, too.
14. If the element e_i must be `ONLY INFORMED OF` e_k and e_k is `AS INTERESTING AS` e_m then e_i must be `ONLY INFORMED OF` e_m, too.
15. If the element e_i must be `INFORMED OF` e_k and e_m is `NOT AS IMPORTANT AS` e_k then e_i must not be `INFORMED OF` e_m.
16. If the element e_i must be `INFORMED OF` e_k and e_k is `NOT AS INTERESTING AS` e_m then e_i must not be `INFORMED OF` e_m.
17. If the element e_i must be `NOT INFORMED OF` e_k and e_m is `NOT AS IMPORTANT AS` e_k then e_i can be `INFORMED OF` e_m.
18. If the element e_i must be `NOT INFORMED OF` e_k and e_k is `NOT AS INTERESTING AS` e_m then e_i can be `INFORMED OF` e_m.
19. If the element e_i must be `ONLY INFORMED OF` e_k and e_m is `NOT AS IMPORTANT AS` e_k then e_i must not be `(ONLY) INFORMED OF` e_m.
20. If the element e_i must be `ONLY INFORMED OF` e_k and e_k is `NOT AS INTERESTING AS` e_m then e_i cannot be `(ONLY) INFORMED OF` e_m.
21. If the element e_i must be `INFORMED OF` e_k and e_m is `ONLY AS IMPORTANT AS` e_k then e_i must be `INFORMED OF` e_m, too.
22. If the element e_i must be `INFORMED OF` e_k and e_k is `ONLY AS INTERESTING AS` e_m then e_i must be `INFORMED OF` e_m, too.
23. If the element e_i must be `NOT INFORMED OF` e_k and e_m is `NOT AS IMPORTANT AS` e_k then e_i can be `INFORMED OF` e_m.
24. If the element e_i must be `NOT INFORMED OF` e_k and e_k is `NOT AS INTERESTING AS` e_m then e_i can be `INFORMED OF` e_m.
25. If the element e_i must be `ONLY INFORMED OF` e_k and e_m is `ONLY AS IMPORTANT AS` e_k then e_i must be `ONLY INFORMED OF` e_m, too.
26. If the element e_i must be `ONLY INFORMED OF` e_k and e_k is `ONLY AS INTERESTING AS` e_m then e_i must be `ONLY INFORMED OF` e_m, too.

4.4.4 Verifying Information-Need CoCons

Dynamic Context Typically, the information needs changes at runtime. An information-need CoCon can refer to context property values in order to select the constrained elements. If their context property values change at runtime then the information-need CoCon can only be checked at runtime. If an information-need CoCon only refers to static context property values that do not change at runtime then the model of the system can be checked already during design.

During Design During design, the question whether some element e_A is informed of some other element e_B boils down to the question whether e_A can access e_B. If e_A must be informed of e_B but cannot access e_B then the model violates the information-need CoCon. Hence, information-need CoCons can be checked via the same mechanism as accessibility CoCons described in section 4.1.4.

At Runtime At runtime, the system must execute the action defined in the CoCon's `ACTION` attribute if e_A and e_B are managed by the system. Hence, the system must monitor its elements. If e_B is changed then it must be checked whether someone is interested in it or not. Furthermore, after modifying e_A it must be checked whether e_A is interested in different elements now. Such monitoring and notification capabilities are topics of research among the information logistics community.

4.4.5 Examples for Using Information-Need CoCons

E-Learning All students who attend a lecture should not read all books in the library, but those books that fit to the topic of the lecture:

```
ALL USERS WHERE 'Role' CONTAINS 'Student'
MUST BE INFORMED OF
ALL DOCUMENTS WHERE 'Topic' INTERSECTS WITH 'User.Lecturetopic'
```

Prefix-Dot-Notation In this example, the prefix-dot-notation explained in section 2.1.10 and 2.2.3 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 refers to the context property values associated with the related element of the 'other' set - the user. The `ACTION` describes when and how to inform the students. It is omitted in this example because its value depends on the software systems used. For example, it could specify 'Instantly Add The Document To The Student's List of Recommended Literature'.

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 INFORMED OF ALL DOCUMENTS WHERE 'Published
in.City' DOES NOT EQUAL 'User.Location.City' OR '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 haven't been published in the city in which the user currently is located. In this example, the target set illustrates how to constrain all users regardless of their context property values via a *total selection* (see section 2.2.1).

Navigation

In this example, the prefix-dot-notation is used in the scope set context condition in oder to navigate. As explained in section 2.1.10, navigation is only possible if an association between the elements is defined. The scope set context condition does not check the values of the context property 'Published in' (associated with a document) and the context property 'Location' (associated with an user). Instead, the values of 'City' associated with the document in the role of 'published in' are checked. Unfortunately, details on using navigation in context conditions are a topic of future research. In section 4.4.2, different ways to answer the question *where?* are discussed. This example demonstrates how to consider both the user's location and the publishing location of the documents when defining information need.

4.5 Value-Binding CoCons

Value-Binding CoCons specify dependencies between context property values.

4.5.1 The Notion of Value-Binding CoCons

Goal: Dependencies between Metadata

Dependencies between context property values can exist as discussed in section 2.1.7.

Constrained Element Types

Both the target set and the scope set of a value-binding CoCon can contain any element type. However, the target set and the scope set must contain elements of only one metaclass, like 'components'. Nothing but 'components' in the target sets or the scope sets of value-binding CoCons are discussed here.

4.5.2 The Value-Binding CoCon Types

The family of value-binding CoCons consists of two CoCon types.

`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: 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

A value-binding can be uni-directional. 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 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, like 'Integrate Two Contracts', belongs to a context property name, like 'Workflow'. One context condition can refer to values of several context property names. The CoCon-type-condition '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 value-binding 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-type-condition ONLY, it must not have any other values for 'Workflow' than 'Integrate Two Contracts' and 'CustomerMarriage'. It is suggested not to use the CoCon-type-condition ONLY for value-binding CoCons because it easily can be misrepresented. Instead, the type-instance constraint explained in section 2.1.4 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 value-binding CoCons:

- The CoCon-type-condition 'ONLY' is not allowed to be used in value-binding 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 value-binding 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 context condition.
- In case of a `FULFILLING THE CONTEXT CONDITION OF` CoCon, the scope set elements must be selected indirectly via one context condition.
- It is not allowed to indirectly select scope set or target set elements via the *unrestricted total selection* `ALL ELEMENTS` (see section 2.2.5). Instead, the valid values definition should be used to describe, which values are allowed to be associated with elements.

Attributes

In case of value-binding CoCons, the `ELSE-ACTION` attribute is more interesting than the `ACTION` attribute because a violation can be automatically repaired. The `ELSE-ACTION` attribute determines what happens if any element violates the CoCon. Two useful values of the attribute `ELSE-ACTION` are suggested for value-binding CoCons: '**Repair Values Automatically**', '**Warn**' or '**Ignore**'.

`ELSE-ACTION` Ignore

The value 'Ignore' of the `ELSE-ACTION` attribute defines that the system ignores the elements that violate a value-binding CoCon. Hence, the illegal elements are filtered, and only legal elements are handled by the system.

`ELSE-ACTION` Warn

The value 'Warn' of the `ELSE-ACTION` attribute defines that the user is informed about detected violations of a the value-binding CoCon. More

precise values of `ELSE-ACTION`, like 'warn system designers via a pop up window and a fire-alarm sound' can be used to define who shall be informed at how the person(s) shall be informed.

`ELSE-ACTION` Repair Values Automatically

The value 'Repair Values Automatically' of the attribute `ELSE-ACTION` defines that the (NOT) missing values are automatically added/removed as follows:

- In case of a `THE SAME AS` CoCon, the context property values associated with an element in the target set are changed in order to fulfil the context condition of the scope set, and the context property values associated with an element in the scope set are changed in order to fulfil the context condition of the target set.
- In case of a `FULFILLING THE CONTEXT CONDITION OF` CoCon, the context property values associated with an element in the target set are changed in order to fulfil the context condition of the scope set.

The algorithm for automatically adding or removing (NOT) missing values is explained in the following section.

4.5.3 The Repair Algorithm for Elements that Violate a Value-Binding CoCon

The 'Other' Context Condition

A CoCon refers to two sets: the target set and the scope set. A value-binding CoCon is violated if an element of one of these sets does (NOT) fulfil the context condition of the **other** set. This 'other' context condition refers to the value(s) $\{v_1^{other}, ..., v_n^{other}\}$ of a context property cp_{other} that are associated with an element e via $values_{cp_{other}}(e)$. The 'other' context condition is not fulfilled because certain values in $\{v_1^{other}, ..., v_n^{other}\}$ are (NOT) associated with e.

Repair Algorithm

The algorithm for automatically fixing elements that violate a value-binding CoCon depends on the comparison condition used in the 'other' context condition. This algorithm is sketched here. It can only work if a *set comparison* condition (see section 2.2.3) is used as condition in the 'other' context condition. Hence, the logical conditions $>$, $\geq$, $<$, or $\leq$ for comparing numbers in the 'other' context condition impede the automatical repair of elements that violate a value-binding CoCon. Moreover, only the following set comparison condition can be automatically repaired:

Adding Values

- If no CoCon-type-condition is used and if the 'other' context condition uses `CONTAINS` as set comparison condition then $\{v_1^{other}, ..., v_n^{other}\}$ are *added* to $values_{cp_{other}}(e)$.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST FULFIL THE CONTEXT CONDITION 'Workflow'` **CONTAINS** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value 'Backup' of the context property 'Workflow' is automatically *added* to all components of which more than 1000 instances exist.

- If the CoCon-type-condition 'NOT' is used and the 'other' context condition uses `DOES NOT CONTAIN` as set comparison condition then $\{v_1^{other}, ..., v_n^{other}\}$ are *added* to $values_{cp_{other}}(e)$.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST NOT FULFIL THE CONTEXT CONDITION 'Workflow'` **DOES NOT CONTAIN** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value 'Backup'

of the context property 'Workflow' is automatically *added* to all components of which more than 1000 instances exist.

Removing Values

- If no CoCon-type-condition is used and if the 'other' context condition uses `DOES NOT CONTAIN` as set comparison condition then $\{v_1^{other}, ..., v_n^{other}\}$ are *removed* from $values_{cp_{other}}(e)$.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST FULFIL THE CONTEXT CONDITION 'Workflow'` **DOES NOT CONTAIN** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value 'Backup' of the context property 'Workflow' is automatically *removed* from all components of which more than 1000 instances exist.

- If the CoCon-type-condition 'NOT' is used and the 'other' context condition uses `CONTAINS` as set comparison condition then $\{v_1^{other}, ..., v_n^{other}\}$ are *removed* from $values_{cp_{other}}(e)$.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST NOT FULFIL THE CONTEXT CONDITION 'Workflow'` **CONTAINS** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value 'Backup' of the context property 'Workflow' is automatically *removed* from all components of which more than 1000 instances exist.

Replacing Values

- If no CoCon-type-condition is used and if the 'other' context condition uses `EQUALS` (=) as set comparison condition then $values_{cp_{other}}(e)$ is set to $\{v_1^{other}, ..., v_n^{other}\}$. Hence, any other value of cp_{other} that was associated with e is *replaced*.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST FULFIL THE CONTEXT CONDITION 'Workflow'` **EQUALS** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value of the context property 'Workflow' is automatically *set* to 'Backup' for all components of which more than 1000 instances exist. Any other value of 'Workflow' that was associated with one of the target set components is removed from this component.

- If the CoCon-type-condition 'NOT' is used and the 'other' context condition uses `DOES NOT EQUAL` (!=) as set comparison condition then $values_{cp_{other}}(e)$ is set to $\{v_1^{other}, ..., v_n^{other}\}$. Hence, any other value of cp_{other} that was associated with e is *replaced*.

  For instance, the value-binding CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST NOT FULFIL THE CONTEXT CONDITION 'Workflow'` **DOES NOT EQUAL** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that the value of the context property 'Workflow' is automatically *set* to 'Backup' for all components of which more than 1000 instances exist. Any other value of 'Workflow' that was associated with one of the target set components is removed from this component.

Problems In three cases, the values cannot be repaired automatically:

- If the 'other' context condition uses `DOES NOT EQUAL` (!=) as set comparison condition then it cannot be repaired automatically.

  For instance, the value-binding CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST FULFIL THE CONTEXT CONDITION 'Workflow'` **DOES NOT EQUAL** `'Backup'`

`WITH ELSE-ACTION = 'Repair Values Automatically'` states that $values_{cp_{other}}(e) \neq$ 'Backup' for all components of which more than 1000 instances exist. However, it cannot be automatically figured out whether other valid values of 'Workflow' should be associated with e or if 'Backup' should be removed from e (leaving $values_{cp_{other}}(e)$ empty). Hence, the 'DOES NOT EQUAL set comparison condition in the 'other' context condition cannot be repaired automatically unless the CoCon's `ELSE-ACTION` attribute defines how to repair it.

- If the CoCon-Type-Condition is 'NOT' the 'other' context condition uses `EQUALS` (=) as set comparison condition then it cannot be repaired automatically.

  For instance, the CoCon `ALL COMPONENTS WHERE '≪Number≫ Amount (in Instances)' > 1000 MUST NOT FULFIL THE CONTEXT CONDITION 'Workflow'` **EQUALS** `'Backup' WITH ELSE-ACTION = 'Repair Values Automatically'` states that $values_{cp_{other}}(e) \neq \{`Backup'\}$ for all components of which more than 1000 instances exist. However, it cannot be automatically figured out whether other valid values of 'Workflow' should be associated with e or if 'Backup' should be removed from e (leaving $values_{cp_{other}}(e)$ empty). Hence, the 'DOES NOT EQUAL set comparison condition in the 'other' context condition cannot be repaired automatically unless the CoCon's `ELSE-ACTION` attribute defines how to repair it.

4.5.4 Detectable Conflicts of Value-Binding CoCons

As explained in section 2.2.2, 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 value-binding CoCon. Please consider that e_{Chico}, e_{Harpo} and $e_{Groucho}$ can represent the same (model) element in different target or scope sets. For instance, the component 'Customer' can both be contained in a scope set (represented via e_{Chico}) and in a target set (represented via e_{Harpo}). 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, like 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 THE SAME AS` e_{Harpo} due to v_1.
2. 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.
3. 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.
4. 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.
5. No element e_{Chico} can be `FULFILLING THE CONTEXT CONDITION OF` e_{Harpo} due to v_1 if e_{Chico} is `NOT FULFILLING THE CONTEXT`

`CONDITION OF` e_{Harpo} due to v_1.

6. Any element $e_{Groucho}$ in the scope set of a `(NOT) THE SAME AS` CoCon must (not) be contained in the CoCon's target set and vice versa.

7. Any element $e_{Groucho}$ in the target set of a `(NOT) FULFILLING THE CONTEXT CONDITION OF` CoCon must (not) fulfil the context condition of the CoCon's scope set.

   item No element e_{Chico} can be `THE SAME AS` e_{Harpo} due to $CoCon_1$ referring 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

4.5.5 Verifying Value-Binding CoCons

Simple Verification

Value-Binding CoCons can be verified more easily than the other CoCon families because only the context property values of their constrained elements must be checked. No additional semantics (e.g. whether one constrained element *accesses* another one as in accessibility CoCons) must be specified or checked.

4.5.6 Examples for Using Value-Binding 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 may be of $VV^{PersonalData}$ = { 'True', 'False'} (see section 2.1.3). 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 2.1.3: an inter-value constraint must prevent contradicting values of one context property for the same element. Value-Binding 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

Prevent Overlapping Domains

In this example, a component called 'Letter Management' exists for managing the company's mail to clients and employees. Amongst others, this component is used in the workflow 'Pay Salary' in order to send out pay checks, and is used by the 'field service' department. A requirement demands that the field service workers must not be involved in the workflow 'Pay Salary'. This can be expressed in CCL via: `ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Pay Salary' MUST NOT BE THE SAME AS ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Field Service'`. The component 'Letter Management' violates this Value-Binding CoCon

because it belongs to both domains. The system (model) must be modified in order to comply with the CoCon. For example, another component responsible for sending out pay checks can be created. Then, the 'Letter Management' component does not belong to the workflow 'Pay Salary' anymore and can be used by the field service.

Setting System Properties

As explained in section 2.1.6, the values of system properties, like 'the current user' or 'the current IP-address', are queried from the middleware platform during configuration or at runtime. A CoCon can refer to system properties in its context condition(s). Value-Binding CoCons can be used to set system property values. For instance, the following CCL statement demands that the 'transaction mode' for each component involved in the workflow 'Pay Salary' is configured as 'Requires a Transaction'. `ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Pay Salary' MUST FULFIL THE CONTEXT CONDITION '≪System ≫ Transaction Mode' = 'Requires a Transaction'`. If any component involved in 'Pay Salary' is not configured to require transactions then the system does not comply with the requirement. The violation of this CoCon can be detected automatically during deployment.

4.6 The Textual Syntax of CCL

Refined CoCon Syntax

A generic syntax for CoCons is defined in section 2.2.5. This section refines this generic CoCon syntax in order to reflect the 21 CoCon types explained in previous sections of this chapter. It defines the syntax of the **C**ontext-Based **C**onstraint **L**anguage CCL. This language consists of all those 21 CoCon types that are introduced in chapter 4. These CoCon types describes requirements within the logical architecture of a component-based system.

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

`CoConType`	::=	**'ACCESSIBLE TO'** | **'READABLE BY'** | **'WRITEABLE BY'** | **'EXECUTEABLE BY'** | **'REMOVEABLE BY'** | **'ALLOCATED TO'** | **'SYNCHONOUSLY REPLICATED TO'** | **'ASYNCHONOUSLY REPLICATED TO'** | **'THE SAME AS'** | **'FULFILLING THE CONTEXT CONDITION OF'** | **'ENCRYPTED WHEN CALLING'** | **'ERRORHANDLED WHEN CALLING'** | **'LOGGED WHEN CALLING'** | **'REDIRECTED WHEN CALLING'** | **'PROTECTED BY A TRANSACTION WHEN CALLING'** | **'ASYNCHRONOUSLY CALLING'** | **'SYNCHRONOUSLY CALLING'** | **'HANDLED WHEN CALLING'** | **'INFORMED OF'** | **'AS INTERESTING AS'** | **'AVAILABLE TO ANYONE INTERESTED IN'**
`Restriction`	::=	**'ELEMENT'** | **'COMPONENT'** | **'CONTAINER'** | **'COMPUTER'** | **'USER'** | **'INTERFACE'**
`Restrictions`	::=	**'ELEMENTS'** | **'COMPONENTS'** | **'CONTAINERS'** | **'COMPUTERS'** | **'USERS'** | **'INTERFACES'**

Keep it Simple

As explained in section 4.5.2, 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 2.2.5:

Syntax of `FULFIL THE CONTEXT CONDITION` CoCons

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

Other CoCon Types

The list of `CoConType`s and `Restriction`(s) listed above is incomplete. It is tailored for one application domain: only requirements for component-based systems are addressed. Moreover, the `Restriction`(s) only consider a few of the metatypes available in the UML components approach. For instance, they do no distinguish between `COMPONENT TYPES` and `COMPONENT INSTANCES`. If other `Restriction`(s) or `CoConType`s are important for expressing requirements then the syntax of CCL can be adapted to new application domains by adding the `Restriction`(s) or `CoConType`s to the BNF rules listed above and by defining additional details as explained in section 2.2.7. Applying CCL to other application domains is not covered here, but it will probably result in additional CoCon types.

5. Conclusion

5.1 Applying CCL in different Levels of the Development Process

Applying CoCons... In this section, the application of CoCons throughout the software development process will be outlined.

...at Requirements Analysis **During requirements analysis**, the business experts must be asked specific questions in order to find out useful context properties and CoCons. Currently, a CoCon-aware method for requirements analysis is being developed at the Technical University of Berlin in cooperation with Eurocontrol, Paris.

...at Desing The application of CoCons during modelling component-based system via UML is currently being evaluated in a case study being carried out in cooperation with the ISST Fraunhofer Institute, the Technical University Berlin and the insurance company Schwäbisch Hall. As proof of concept, the 'CCL plugin' for the open source CASE tool ArgoUML has been implemented and is available for download at `ccl-plugin.berlios.de/`. It integrates the verification of CoCons into the Design Critiques ([51]) mechanism of ArgoUML. Hence, it demonstrates how to verify UML models for compliance with CoCons. In order to reveal conflicting requirements and to detect problems early on, they should be written down in models. Fixing them during implementation is much more expensive. The benefits of considering both requirements and architecture throughout the modelling level are discussed in [42].

...at Deployment **During configuration**, a CoCon-aware configuration file checker can automatically protect requirements. For instance, deployment descriptors of Enterprise Java Bean Systems can be checked for compliance with CoCons. Likewise, the notion of contextual diagrams is introduced in [24] in order to cope with the intrinsic complexity of configuration knowledge.

...at Runtime Usually, the source code of components is modified in order to adapt a component-based system to new requirements. Sometimes, however, it can be impossible to modify autonomous parts of the system. A framework for intercepting communication calls in order to monitor them for compliance with CoCons is described in [13, 35]. It facilitates *not* to implement new requirements *into* any of the components involved. Instead, conformity with requirements is enforced *externally* of the components. Context properties are associated with components externally, and the communication calls between the components are monitored for whether the current call violates a CoCon or not. Hence, the system can be *transparently* adapted to new requirements without modifying the components directly. The framework has been prototypical integrated into an application server at the TU of Berlin in cooperation with the Fraunhofer ISST and BEA Systems ([6, 59]). Thus, legacy components or 'off the shelf' components can be forced to comply with new requirements at runtime.

5.2 Comparing OCL to Context-Based Constraints

OCL Typically, the *Object Constraint Language OCL* ([45, 60]) is used for

the constraint specification of UML models. This section compares OCL with CoCons.

5.2.1 New: Indirectly Constrained Elements

No Indirect Selection of Elements

One OCL constraint is associated with (normally one) *directly identified* model element, while a context-based constraint can refer to both directly identified and to (normally many) indirectly identified, *anonymous and possibly unrelated* elements. A CoCon can select the constrained elements according to their metadata. Context properties are similar to tagged values in UML - on the design level, tagged values can be used to express context properties. The concept of selecting the constrained elements of a constraint via their tagged values is unknown in OCL or in any other existing constraint language.

Considering Unknown Elements

The key new concept of CoCons is the indirect selection of constrained elements. When specifying an OCL constraint, it is not possible to consider elements that are unknown at specification time. Instead, any unknown element becomes involved in a context-based constraint simply by having the matching context property value(s). Hence, the constrained model elements can change without modifying the CoCon specification. The indirect selection of constrained elements is particularly helpful in highly dynamic systems or models. Every new, changed or removed (model) element is automatically constrained by a CoCon due to the element's context property values.

Considering Possibly Unrelated Elements

An OCL constraint can only refer to elements that are directly linked to its scope (called 'context' in OCL according to [18]) via 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 or systems.

CoCons are not like 'Yellow Sticky Notes'

In section 1.2, the metaphor 'yellow sticky note' is used for explaining context properties. Context property values stick to a model element and describe the context of this element. Likewise, OCL constraints stick to a model element. They constrain the instances of this model element as explained in the next section. CoCons, however, may *not* stick to a model element. Instead, one CoCon can *indirectly* select many constrained elements via the context property values sticking to the elements. A CoCon can be written on a yellow sticky note if this CoCon *directly* selects the model element to which this yellow note sticks. In that case, it can refer to this directly selected element via the keyword THIS (see section 2.2.5). A CoCon should not be written on a yellow sticky note of an element if this CoCon indirectly selects the element via its context property values because this CoCon might not constrain the element anymore if the element's context property values change. As well, the same CoCon can constrain other elements if their values match the CoCon's context condition. Should one CoCon be written on many sticky notes that stick to each indirectly constrained element? If yes then all these sticky notes are redundant because they all contain the same CoCon. When changing the CoCon, all sticky notes on which is was written must be adapted. In order to keep it simple and consistent, a CoCon should not be attached to indirectly constrained elements. CoCons can *indirectly* select the constrained elements via the 'yellow sticky notes' attached to the elements, but CoCons may not be *directly* attached to the constrained elements like 'yellow sticky notes'.

5.2.2 New: Verifying CoCons already during Design

OMG Meta-Levels

The Object Management Group (OMG) describes four metal-levels: Level 'M_0' refers to a system's objects at runtime, 'M_1' refers to a system's model or schema, such as a UML model, 'M_2' refers to a metamodel, such as the UML metamodel, and 'M_3' refers to a meta-metamodel, such as the Meta-Object Facility (MOF) . Note that level M_k elements are 'instances' of level M_{k+1} elements.

CoCons Can be Verified on the Same Meta-Level

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' ([18]) of an invariant is an *instance* of the associated model element. If an OCL constraint specifies the range of values for an attribute of a model element on M_1 level, the *current* value of this attribute must comply with the constraint on level M_0 *at runtime*. The value of an attribute on M_1 level is unknown during design. In order to check the compliance of a model with a M_1 level OCL constraint, either M_0 level instances must be simulated as proposed in [50], or the OCL constraint must be translated into source code as in [30] in order to check the constrained values at M_0 level. On the contrary, context property values are already defined in the model. Hence, they can be verified on the *same* meta-level where the CoCon is specified because the values referred to by the CoCon are defined on the same metalevel as the constraint. The model's compliance with a CoCon can be checked according to the CoCon type on the same metalevel where the context property values are given as explained in chapter 4.

5.2.3 Mapping CoCons to OCL

Example

This section discusses only one example: As illustrated via the dependency relationship (the dotted arrow) in the UML deployment diagram shown in figure 5.1, component 'A' invokes component 'B'. According to the following M_1 level CoCon, the component 'A' is not allowed to the component invoke 'B', though:

..in CCL

```
ALL COMPONENTS WHERE 'Personal Data' = 'True'
MUST NOT BE ACCESSIBLE TO
ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'
```

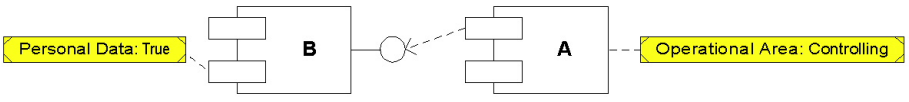

Figure 5.1: The Component 'A' is Not Allowed to Access The Component 'B'

..in OCL

This CoCon can incompletely be expressed in OCL on M_2 (!) level of UML 1.4 as:

```
context  component inv:
     if  self.taggedvalue->select(tv | tv.dataValue =
         "Controlling")
         .type -> select(td | td.name = "Operational Area")
         -> notEmpty()
    and  self.clientDependency.supplier
         -> select(i | i.oclIsTypeOf(Interface)
         .clientDependency
         -> select(d | d.oclIsKindOf(Abstraction)
                     and d.stereotype.name = "realize")
         .supplier -> select(c | c.oclIsTypeOf(Component)
         .taggedvalue -> select(tv | tv.dataValue = "True")
         .type -> select(td | td.name = "Personal Data")
         -> Empty()
```

Incomplete OCL Statement

The OCL constraint specified here is incomplete. It would be much longer if it would cover the following details. First, it only defines inaccessibility for component types, but not for component instances. Moreover, it does not consider messages between components in a sequence diagram by stating that the operations in an interface of an inaccessible component cannot be invoked by a component belonging to the target set. Additionally, other diagram types that can specify communication between components, like collaboration diagrams, state or activity diagram, are not considered in the OCL statement given above. For each of these diagrams, the concept of (in)accessability must also be reflected in the OCL statement. Furthermore, it does not handle recursion ([17]): if the component 'B' in figure 5.1 does not manage personal data, but invokes another component 'C' handling personal data, than 'A' must not invoke an operation of an interface of 'B' as explained in section 4.1.2. As well, 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 described in chapter 3 specifies components via stereotyped classes (see figure 3.2). The incomplete OCL statement given above does not consider stereotyped classes.

OCL Statement Too Precise

The OCL expression is much longer than the CCL expression because describes a lot more details. It precisely defines the meaning of inaccessibility. However, the precise description of 'inaccessibility' differs at the various metalevels. Each precise constraint on each metalevel must be adapted if the corresponding requirement changes. The meaning of 'inaccessibility' is not defined in the constraint itself here. Instead, the accurate semantic of 'inaccessibility' can be provided either by translation templates for an `ACCESSIBLE TO` CoCon into each metalevel or by a tool that checks the system for compliance with the requirement at the particular metalevel according to its own definition of 'inaccessibility'. Hence, the same abstract CoCon can be checked on different metalevels.

Incomplete UML

Not every CoCon type can be mapped easily to UML because UML may lack of the necessary concepts. For instance, a communication CoCon can demands that the communication with a component in its target set must be encrypted. In standard UML, there is no concept for specifying whether communication is encrypted or not. It can be provided by an UML profile. However, without such a profile it is not possible to express certain CoCon types in OCL. When using a certain profile, the OCL expressions must be adapted to this profile if it introduces a new or different concepts. The profile may, e.g., define a new concept

for specifying accesability. This concept must be considered in all OCL expressions that express accessibility requirements. An accessibility CoCon, however, must not be adapted to a profile because it is independent of how communication calls between components are specified.

The Only Constant in Live is Change

Keeping track of requirements like sitting on a cat that's playing with a mouse – the requirements always keep changing. The effort of writing down a requirement in the minutest details may be unsuitable if the details are not important. CoCons facilitate staying on an abstract level that eases requirement specification. Moreover, CoCons facilitate checking the *model* for compliance with requirements. CoCons can be specified and validated on the same metalevel as the context property values to which they refer. On the contrary, an OCL constraint expressing the same requirement as a CoCon must be specified on M_2 level in order to check it on M_1 level. Modifying the *meta*model each time the requirements change is not appropriate.

5.2.4 'Agile Processes' need Invariants

Agile Processes

Agile Processes[1] (AP) are emerging as a new, lightweight methodology for software engineering. Their goal is to adopt and extend the practices of eXtreme Programming (XP) to other abstraction levels, such as modelling. Up to now, AP practices are still in their infancy. Nevertheless, the success of XP should encourage us to reconsider design processes accordingly.

Unit Tests

One of the outstanding practices of XP is testing, where invariants called 'unit tests' check the source code repeatedly in order to provide confidence in the developed system, as well as the flexibility to change it. According to [7], the principles of XP should also be applied in models. As described above, the compliance of a model with CoCons can already be checked at the modelling level. Hence, CoCons are invariants that enable the testing of models in the spirit of unit tests.

Design Rationale

There used to be a lot of interest in recording the *design rationale*. According to [41], the idea was that designers would not only record the results of their design thinking, but also the reasons behind their decision. Thus, they would also record their justification for why it is as it is. CoCons record design decisions that can be automatically checked. They represent certain relevant requirements in the model. The problem is that designers simply don't like writing down design decisions. The challenge is to make the effort of recording the rationale worthwhile and not too tedious for the designer. As a reward for writing down design decisions via CoCons they reap the benefits summarized in section 5.4.

5.3 Limitations of CCL

Trustworthy Metadata

Taking only the meta data of an element into account bears some risks. It must be ensured that the context property values are always up-to date. The following approaches can improve the quality of context property values:

- Metadata can be automatically extracted from its element. A survey on techniques for extracting metadata is given in [31]. If the metadata is extracted newly each time when checked and if the extraction mechanism works correctly then the metadata is correct

[1] 'Agile Processes' are also known as 'Agile Modelling' or 'eXtreme Modelling'

and up-to-date. Moreover, the extraction mechanism ensures that metadata is available.

- Contradicting context property values can automatically be prevented via inter-value constraints as explained in section 4.5.2.
- Additional Metadata can be automatically derived from existing metadata as explained in section 4.5.3.
- Whoever holds the responsibility for the values must be trustworthy. Confidence can be assisted with security techniques.
- A policy should be enforced that demands to annotate each added or changed element with those context properties that are referred by the CoCons defined for the system.

Example If, e.g., the workflow 'Integrate Two Contracts' is part of the workflow 'CustomerMarriage' and a CoCon is specified for all components that are needed to execute the workflow 'CustomerMarriage' then the value 'CustomerMarriage' should be assigned to all components which already have the value 'Integrate Two Contracts' for 'Workflow'. Otherwise, some components, which should be constrained by this CoCon, are ignored because they are only associated with the value 'CustomerMarriage', but not associated with the value 'Integrate Two Contracts'. Value-Binding CoCons can enforce such dependencies between context property values. Hence, they facilitate managing the metadata and keeping it consistent.

Missing Metadata Some elements may not be annotated with the relevant metadata. A mechanism that checks the system for compliance with CoCons should be able to cope with missing metadata – it should have a strategie for handling elements without metadata. For example, strategies for handling elements without metadata in invocation paths when checking accessibility CoCons are discussed in [59].

Ontologies Within one system, only one ontology for metadata should be used. For instance, the workflow 'New Customer' should have exactly this name in every part of the system, even if different companies manufacture or use its parts. Otherwise, string matching gets complex when checking a context condition. If more than one ontology for metadata is used, correspondences between heterogeneous context property values can be expressed via constraint or correspondence techniques, like value-binding CoCons or Model Correspondence Assertions ([14]).

Science Fiction Some of the concepts that can be expressed via CoCons are not supported by middleware platforms nowadays. 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. Nevertheless, dynamically changing transaction modes are not supported by middleware platforms yet.

Only Dual In order to keep it simple, the CoCon types defined here only refer to two sets of elements: the target set and the scope set. One CoCon cannot express a relationship between three or more sets of elements. It is a topic of future research to discuss, which n-ary relationships between n sets of elements can or should be transformed into binary CoCons.

Conflicts Hardly Detectable Without Constrained Elements In chapter 3, automatically detectable conflicts are listed for each CoCon family. These conflicts, though, can hardly be detected if no constrained elements exist because it is unknown which elements will be selected by the CoCon's context condition(s). If no constrained elements exist

then only the CoCon statements themselves can be analysed. Conflicts between CoCon *statements* can only be detected if both their scope set selection and their target set selections are identical. In that rare case, the same elements will be selected by the identical set selections. CoCons with identical set selections will always constrain the same elements. For example, a conflict between the following two CoCons can be detected without knowing their constrained elements because the set selections of both the scope set and the target set are identical:

1. `ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes' MUST` **`NOT`** `BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'`
2. `ALL COMPONENTS WHERE 'Personal Data' EQUALS 'Yes' MUST BE ACCESSIBLE TO ALL COMPONENTS WHERE 'Operational Area' CONTAINS 'Controlling'`

In general, the CoCons of one system will have different scope set and target set selections. Hence, conflicts between them can only be detected if they constrain any elements.

EJB Limitation

As explained in section 4.1.5 and in [6, 59], not all CoCons can transparently be monitored in Enterprise Java Beans (EJB) systems at runtime because an EJB component cannot identify the caller of its interface. The context property values of the calling component cannot be determined without modifying the source code of the components. The current user, however, can transparently be identified in EJB systems at runtime. Therefore, the scope set of a CoCon should refer to users instead of components if the CoCon shall be applied to EJB systems at runtime. An example is given in section 4.1.5.

5.4 Benefits of CCL

Context Properties

In contrast to other grouping techniques, e.g. packages or stereotypes, context properties properties can dynamically group elements at runtime. They facilitate handling of overlapping or varying groups of elements that share a context even across different element types or diagrams. Hence, One requirement affecting several components that do not invoke each other directly or even do not run on the same platform can now be expressed via one constraint. Context properties allow *subject-specific, problem-oriented views* to be concentrated on. For instance, only those elements belonging to workflow 'X' may be of interest in a design decision. Many concepts for specifying metadata exist and can be used instead if they enable a constraint to select the constrained elements via their metadata. Similar to context properties, [46] suggest to assign different kinds metadata to components in order to use it for different tasks throughout the software engineering lifecycle. Likewise, [56] annotate components in order to perform dependence analysis over these descriptions. Other concepts for considering metadata exist, but none of them writes down constrains that reflect this metadata as introduced here.

Comprehensibility

The model should serve as a document understood by designers, programmers and customers. CoCons can be specified in easily comprehensible, straightforward language. As described in section 5.2, most CoCon statements can be translated into the standard UML constraint language OCL. The translated OCL statement is much longer and much more complicated than the CCL statement, though. The effort of writing down a requirement in the minutest details is unsuitable if the details are

not important. CoCons facilitate staying on an abstract level that eases requirement specification.

Coping with Complex Systems

The design of a software system must satisfy the requirements while not violating constraints imposed on the problem domain and implementation technologies. However, in complex domains, no one architect has all the knowledge needed to design a complex system. Instead, most complex systems are designed by teams of stakeholders providing some of the necessary knowledge and their own goals and priorities. This 'thin spread of application domain knowledge' has been identified by [19] as a general problem in software development. In complex domains even experienced architects need knowledge support. For instance, they need to be reminded which of the requirements apply to which part of the system. Even the person who specifies a requirement via CoCons does not have to have the complete knowledge of the system due to the *indirect* association of CoCons to the system parts involved. It can be unknown which components are involved in the requirement when writing down the CoCon. CoCons associate relevant requirements with the related components.

Reasoning in the Presence of Inconsistency

In software engineering, it has long been recognized that inconsistency is a fact of life. Evolving descriptions of software artefacts are frequently inconsistent, and tolerating this inconsistency is important if flexible collaborative working is to be supported. The abstract metadata belonging to an element can be determined in an early lifecycle activity. When identifying the context property values the element must not be specified in full detail. Metadata can supplement missing data based on experience or estimates. The people who need a new requirement to be enforced often neither know the details of every part of the system nor do they have access to the complete source code. By using CoCons, developers don't have to understand every detail ('glass box view'). Instead, they only must understand the metadata. CoCons support the design of software systems from the start of the development process. They facilitate detecting problems early during modelling by identifying conflicting requirements and assist in establishing a trade off.

Evolution

Maintenance is a key issue in *continuous software engineering* (see section 1.1). CoCons help to ensure consistency during system evolution. A context-based constraint serves as an invariant and, thus, prevents the violation of design decisions during later modifications of the model or the system. It assists in detecting when design or context modifications compromise intended functionality. It helps to prevent unanticipated side effects during redesign and it supports collaborative design management. Requirements tend to change quite often. Indirection improves adaptability — the constrained elements can be indirectly selected according to their metadata. This metadata can be easily adapted whenever the context of a component changes. Furthermore, each deleted, modified or additional CoCon can be automatically enforced and any resulting conflicts can be identified as discussed in chapter 4. 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.

Validation at Modelling / at Configuration / at Runtime

CoCons can be verified during modelling, during deployment and at runtime. They facilitate description, comprehension and reasoning on different levels and support checking the compliance of a system with requirements automatically. According to [39], automated support for software evolution is central to solving some very important technical problems in current day software engineering.

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.

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

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

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

[5] 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.

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

[7] Marko Boger, Toby Baier, Frank Wienberg, and Winfried Lamersdorf. Structuring QoS-supporting services with smart proxies. In *Extreme Programming and Flexible Processes in Software Engineering - XP2000*, Reading, 2000. Addison-Wesley.

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] Cory Casanave. Business-object architectures and standards. In J. Sutherland, D. Patel, C. Casanave, G. Hollowell, and J. Miller, editors, *OOPSLA Workshop on Business Object Design and Implementation*. Springer, 1995.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] Desmond D'Souza and Alan Cameron Wills. *Catalysis – rigorous component-based development*. 2000.

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

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

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

[26] Martin Fowler and Kendall Scott. *UML Distilled*. Object Technologies. Addison-Wesley, Reading, 1999.

[27] Joseph Gil, John Howse, and Stuart Kent. Constraint diagrams: A step beyond UML. In *TOOLS, USA*. IEEE Computer Society Press, December 1999.

[28] Olly Gotel and Anthony Finkelstein. An analysis of the requirements traceability problem. In Robert Arnold and Shawn Bohner, editors,

Software Change Impact Analysis. IEEE Computer Society Press, 1996.

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

[30] 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.

[31] Sam Joseph. Decentralized meta-data strategies. University of Tokyo, draft available at www.neurogrid.net/Decentralized_Meta-Data_Strategies-neat.html, 2002.

[32] 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.

[33] Philippe Kruchten. Modeling component systems with the unified modeling language. In *International Workshop on Component-Based Software Engineering*. Carnegie Mellon University, 1998.

[34] Ralf-Detlef Kutsche and Asuman Sünbül. A meta-data based development strategy for heterogeneous, distributed information systems. In *3rd IEEE Metadata Conference, Bethesda, Maryland*, April 1999.

[35] 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.

[36] Information Logistics. www.informationslogistik.org. Last visit in July 2002.

[37] Stefan Mann, Alexander Borusan, Hartmut Ehrig, Martin Große-Rhode, Rainer Mackenthun, Asuman Sünbül, and Herbert Weber. Towards a component concept for continuous engineering. Internal Report of the BMBF Research Project *Continuous Engineering*, 2000.

[38] Nenad Medvidovic. A classification and comparison framework for software architecture description languages, uci-ics-97-02. Technical report, University of California, Irvine, February 1997.

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

[40] Miguel de Miguel, Thomas Lambolais, Sophie Piekarec, Stéphane Betgé-Brezetz, and Jérôme Pequery. Automatic generation of simulation models for the evaluation of performance and reliability of architectures specified in UML. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, volume 1999 of *LNCS*, pages 82–100, Berlin, 2000. Springer.

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

[42] Bashar Nuseibeh. Weaving the software development process between requirements and architecture. In *Proceedings of ICSE-2001 International Workshop: From Software Requirements to Architectures (STRAW-01) Toronto, Canada*, 2001.

[43] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: A roadmap. In *Proceedings of International Conference on Software Engineering (ICSE2000), Limerick, Ireland*. ACM Press, June 2000.

[44] OMG. UML specification v1.3. OMG-Document ad/99-06-08, 1999.

[45] OMG. UML specification v1.4, September 2001.

[46] Alessandro Orso, Mary Jean Harrold, and David Rosenblum. Component metadata for software engineering tasks. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, volume 1999 of *LNCS*, Berlin, November 2000. Springer.

[47] Joseph P. Pickett, editor. *The American Heritage Dictionary of the English Language*. Boston: Houghton Mifflin Company, 2000.

[48] Matthias Radestock and Susan Eisenbach. Semantics of a higher-order coordination language. In *Coordination 96*, 1996.

[49] Paul Resnik and Hal R. Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[50] 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.

[51] 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.

[52] R. Schmidt and U. Assmann. Concepts for developing component-based systems. In *International Workshop on Component-Based Software Engineering*. Carnegie Mellon University, 1998.

[53] 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.

[54] Oliver Sims. *Business Objects: Delivering Cooperative Objects for Client-Server*. McGraw-Hill, 1994.

[55] Martin Skinner. Enhancing a UML editor by context-based constraints for components. Diploma Thesis at the Technical University Berlin, October 2001.

[56] Judith A. Stafford and Alexander L. Wolf. Annotating components to support component-based static analyses of software systems. In *Grace Hopper Celebration of Women in Computing, Hyannis, Massachusetts*, September 2000.

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

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

[59] Jianxin Wang. Prototypische Entwicklung eines Compilers zur Umwandlung von Context-Based Constraints in ECA-Regeln. Diploma Thesis at the Technical University Berlin, Research Group CIS, June 2002.

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

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

[62] Herbert Weber. Continuous engineering of information and communication infrastructures (extended abstract). In Jean-Pierre Finance, editor, *Fundamental Approaches to Software Engineering FASE'99 Amsterdam Proceedings*, volume 1577 of *LNCS*, pages 22–29, Berlin, March 22-28 1999. Springer.

[63] Alan Cameron Wills. Designing component kits and architectures with catalysis. In Leonor Barroca, editor, *Software architectures: advances and applications*. Springer, 1999.

[64] J. Leon Zhao, Akhil Kumar, and Edward A. Stohr. A dynamic grouping technique for distributing codified-knowledge in large organizations. In 10^{th} *Workshop on Information Technology and Systems, Brisbane, Australia*, December 2000.