What Must (Not) Be Available Where?

Felix Bübl[1]

imphar AG, Berlin, Germany
felix.buebl@imphar.com

Abstract Arranging the distribution of data, objects or components is a critical task that can ultimately affect the performance, integrity and reliability of distributed system. This paper suggests to write down *what must (not) be available where* in order to reveal conflicting distribution requirements and to detect problems early on. Distribution requirements are expressed via a new notion of constraints: a context-based constraint (CoCon) can indirectly select its constrained elements according to their context. The context of an element characterizes the situation in which this element resides and is annotated via metadata. CoCons facilitate checking the compliance of a system or a model with distribution requirements during (re-)design, during (re-)configuration or at runtime. This paper focuses on validating UML models for compliance with distribution CoCons in order to take distribution requirements into account right from start of the development process.

1 Introduction

1.1 Recording Distribution Decisions

Up to now the rationale for distribution decisions is barely recorded during the design and development of software system. Instead, distribution is typically taken into account during implementation or deployment and is expressed directly in configuration files or in source code. But, the context for which a software system was designed changes continuously throughout its lifetime. In order to prevent the violation of distribution requirements when adapting a systems to new requirements, the distribution requirements must be written down. By using a formal language for expressing distribution requirements, a system can automatically be checked for whether it complies with these distribution requirements.

After distributed applications became popular and sophisticated in the 80s, over 100 *programming languages* specifically for *implementing* distributed applications were invented according to [2]. But hardly anyone took distribution into consideration already on the *design* level. In order to reveal conflicting distribution decisions and to detect problems early on, they should be written down and considered already in the model. Fixing them during implementation is much more expensive. Moreover, distribution requirements should be expressed in an *artefact-independent way*: it should be possible to check the system's *model* as

well as its *source code* or its *configuration files* for compliance with distribution requirements without restating the distribution requirements for each artefact type. Artefact-independent expressions can be considered throughout the lifetime of a distributed system – they enable software development tools to detect the violation of distribution requirements automatically during (re-)design, (re-) configuration or at runtime.

This paper proposes to specify distribution requirements via constraints. For instance, one distribution constraint can express which element must (not) be allocated to which computer. An element can be an *object*, *data*, or *component*. This paper focuses on components. When applying the approach discussed here during modelling, please read 'element' as *model element* throughout the paper. However, keeping track of each individual elements becomes increasingly difficult if the number of components or computers grows. In large-scale systems, it is not practical to specify distribution constraint relating to individual components or individual computers. Instead, it must be possible to specify constraints relating to possibly large groups of elements. Furthermore, this difficulty increases if the components or computers involved change frequently. In complex or frequently changing system, it is too expensive to write down which individual component must (not) reside on which individual computer. Therefore, this paper suggest to express distribution requirements in an *adaptive* way: a new specification technique is presented that defines what must (not) be available where according to the components's or the computer's context.

1.2 Example: Availability Requirement

The following availability requirement is used as an example in this paper: *All components needed in the workflow 'Create Report' must be allocated to all computers belonging to the 'Controlling' department.* Due to this requirement, any computer of the controlling department can access all the components needed in the workflow Create Report even if the network fails. Hence, the availability of the workflow Create Report is ensured on these computers. But, which system elements should be checked for whether they are (not) allocated to which computers? How can the system be checked for compliance with this requirement? The answers start in the next section.

1.3 The Paper in Brief

This paper suggests defining distribution requirements via Context-Based Constraints (CoCons). Their basic concept introduced in [5] can be explained in just a few sentences.

1. The system elements, e.g. components or computers, are annotated with formatted metadata called 'context properties'. A context property describes its element's context. As defined in section 2, context is any information that can be used to characterize the situation of an element.

2. Only those elements whose context property values fit the CoCon's context condition must fulfil the constraint. Up to now, constraints do not indirectly select their constrained elements according to their context properties as explained in section 3.
3. A CoCons refers to *two* sets of elements. It relates each element x of one set to each element y of the other set and defines a condition $C(x, y)$ on each pair of related elements. If one set contains components and the other set contains computers then this condition on a pair of related elements can state that 'x must be allocated to y' as discussed in section 4.

When adapting a systems to new, altered or deleted requirements, existing distribution requirements should not unintentionally be violated. Two different kinds of automatically detectable CoCon violations are discussed in this paper:

- An **inter-CoCon conflict** occurs if one CoCon contradicts another CoCon as discussed in section 5.
- An **illegal artefact element conflict** occurs if a artefact element does not comply with a CoCon's predicate on how it must (not) relate to another artefact element. As an example, checking the compliance of UML deployment diagrams with distribution requirements is discussed in section 6.

2 Context Properties

2.1 What is Context?

This paper focuses on the context of software system elements. It uses context for one specific purpose explained in previous section: it refers to context of software system elements in order to distinguish those elements that reside in the same context from other elements that don't.

The context models used in software engineering typically focus on internal context of software systems as explained next. A software system consists of artefacts, like source code files, configuration files, or models. One artefact can consist of several elements. An **internal element** is contained in at least one of the system's artefacts. For example, the name of a component, the name of a method, or the name of a method's parameter are internal elements. On the contrary, an **external element** is not contained in any of the system's artefacts. An **internal context** of a software system element refers to other internal elements. It does not refer to external elements.

For example, the 'context of a component' is defined as 'the required interfaces and the acceptable execution platforms' of components in [23]. This is an internal notion of context because it only refers to internal elements: other components or containers are defined as context of a component. Likewise, the context of an Enterprise Java Bean is defined as 'an object that allows an enterprise bean (EJB) to invoke container services and to obtain information about the caller of a client-invoked method'. Once more, the context of a component

(the EJB) only refers to other internal elements: both the container and the calling component are system elements. This paper proposes also to take external contexts into account. It suggests to select constrained elements according to their context regardless whether their context is part of the system or not.

Context is defined in [11] as 'any information that can be used to characterize the situation of an entity'. This definition needs a precise definition of 'situation'. In situation calculus ([10]), **situation** is defined as structured part of the reality that an agent manages to pick out and/or to individuate. This definition suits well for this paper because context is used here for distinguishing those elements that are involved in a requirement from the other elements that don't. A context is not a situation, for a situation (of situation calculus) is the complete state of the world at a given instant. A single context, however, is necessarily partial and approximate. It cannot *completely* define the situations. Instead, it only characterizes the situation.

Section 2.2 will present a syntax and informal semantics for expressing (situational) context of software elements.

2.2 Context Properties: Formatted Metadata Describing Elements

The context of an element can be expressed as metadata. 'Metadata' is typically defined as 'data about data'. According to [1], the attribute-value pair model is the commonly used format for defining metadata today. As well, the context of a system element is epressed in the simple attribute-value syntax here: a **context property** consists of a name and a set of values.

First, this section defines a textual syntax of context properties via BNF rules. Afterwards, it informally explains the semantics of context properties.

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. The following syntax is used for assigning values of one context property to an element:

```
ConPropValues ::= ContextPropertyName [‘(’ ElementName ‘)’ ] ‘:’ Context-
   PropertyValue {‘,’ ContextPropertyValue}
```

A context property **name** (called `ContextPropertyName` in the syntax definition given above) groups context property values.

For instance, the values of the context property 'Workflow' reflect the most frequent workflows in which the associated element is used. Only the names of the workflows used most often are taken into account for requirement specification here. In this case, the name of the context property is 'Workflow'. The BNF rule `ContextPropertyValue` defines the **valid values** of one context property name. For instance, the four values allowed for Workflow can be `ContextPropertyValue := ‘New Contract’ | ‘Delete Contract’ | ‘Create Report’ | ‘Split Contract’`. A subset of the valid values can be associated

with a single element for each context property name. These values describe how or where this element is used – they describe the context (as discussed in section 2.1) of this element. The context property name stays the same when associating its values with several elements, while its values might vary for each element.

2.3 Research Related To Context Properties

Many notations for writing down metadata as attribute-value pairs exist and can be used for expressing the context of elements. For example, tagged values ([15]) can be used to express context properties in UML.

As summarized in [20], a concept similar to context properties was discussed in the 90ties: database objects are annotated via intensional description called 'semantic values' ([21,19]) in order to identify those objects in different databases that are semantically related. Likewise, context properties are annotated to elements in order to determine the relevant element(s). However, the semantic value approach has a different purpose and, thus, a different notion of *relevant*: the purpose of semantic values is to identify semantically related objects in order to resolve schema heterogeneity among them. On the contrary, context properties are not annotated in order to identify those database objects in heterogeneous databases that correspond to each other. Instead, context properties are annotated to elements in order to identify those elements that are involved in a certain requirement as explained in the next section. The application of semantical values differs from the application of context properties. Still, the concepts are similar because they both can denote elements residing in the same context.

Another concept similar to context properties are 'domains' introduced in [22]. They provide a means of grouping objects to which policies apply. In contrast to context properties, a domain does not consist of a name and corresponding values. Instead, a domain consists of one single term. Domains are a unit of management similar to file directories in operating systems, and provide hierarchical structuring of objects. As well as any other metadata concept, domains can be used to express the context of elements. Section 3.1 will explain that context can be expressed with any metadata concept as long as a query language for the other metadata concept exists.

A context property groups elements that share a context. Existing grouping mechanisms like inheritance, stereotypes or packages are not used because the values of a context property associated with one element might vary in different configurations or even change at runtime. An element is not supposed to change its stereotype, its inheritance, or its package at runtime. Context properties are a simple mechanism for grouping otherwise possibly unassociated elements - even across different views, artefact types, or platforms. The primary benefit of enriching elements with context properties is revealed in the next section, where they assist in identifying those elements that are involved in a requirement.

3 Context-Based Constraints (CoCons)

3.1 Indirect Selection of Constrained Elements

This section presents a new notion constraints that can indirectly select the constrained elements when expressing which component must (not) be available at which computers. A **context-based constraint** (CoCon) can indirectly select the constrained elements according to their context property values. It expresses a condition on how its constrained elements must be related to each other. This condition is called **CoCon-predicate** here. Different types of CoCon-predicates exist. A 'CoCon-predicate type' is abbreviated as **CoCon type** here. Hence, 'CoCon type' is a synonym for 'CoCon-predicate'. The **C**ontext-Based **C**onstraint **L**anguage **CCL** introduced in [4] consists of 21 different types of CoCons. This paper, however, discusses only those CoCon types of CCL that define distribution requirements.

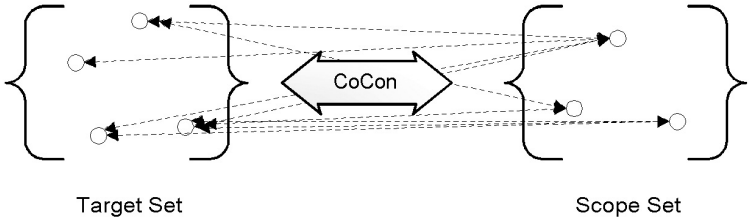

Figure 1. A CoCon Relates any Element of the 'Target Set' with any Element of the 'Scope Set'

Figure 1 illustrates that a CoCon relates each element of one set to each element of the other set and expresses a CoCon-predicate (depicted as dotted arrows) for each *pair* of related elements. The two sets related by a CoCon are called 'target set' and 'scope set'. Both target set elements and scope set elements of a CoCon can either directly or indirectly be selected. Indirect selection is the key concept of context-based constraints. A CoCon can *indirectly* select set elements via a **context condition** that defines, which context property values an element must (not) have in order to be constrained by the CoCon. If the context property values of an element comply with the CoCon's context condition then the CoCon constrains this element: in that case, this element must fulfil the CoCon-predicate in relation to other constrained elements. In section 1.2, the target set elements are selected via the following context condition: "*All components whose context property 'Workflow' has the value 'Create Report'* ". These target set elements are anonymous. They are not directly named or associated, but described indirectly according to their context property values. If no element fulfils the context condition, the set is empty. This simply means that the CoCon actually does not apply to any element at all.

The same set of elements can both be selected directly and indirectly. For example, the direct selection '$component_1$, $component_4$ and $component_7$' can describe the same components as *All components whose context property 'Workflow' has the value 'Create Report'*. However, the indirect selection *automatically* adapts to changed elements or context changes, while the direct selection doesn't. For instance, eventually a new component will be managed by the system that was not managed yet when writing down the CoCon. The new component $component_{31}$ is not selected by the direct selection given above. On the contrary, the indirect selection will automatically apply to $component_{31}$ as soon as $component_{31}$'s context property 'Workflow' has the value 'Create Contract'. The indirect selection statement must not be adapted if system elements or their contexts change. Instead, the indirectly selected elements are identified by evaluating the context condition each time when the system is checked for whether it complies with the CoCon.

The concept of CoCons is independent of the *context property data schema*. A simple and flat attribute-value schema for context properties has been introduced in section 2. Of course, more expressive data schemata for storing the context properties of one element, e.g. hierarchical, relational, or object-oriented schemata, can be used. The *query language* used to specify the context condition depends on the context properties data schema. If the context properties of each artefact element are stored in a relational schema then a relational query language, e.g. SQL, can be used to express context conditions. If the context properties are, e.g., stored in a hierarchical XML schema then a query language for XML document can be used, e.g. XQuery. However, this paper focuses on the non-hierarchical, non-relational, non-object-oriented data schema for context properties defined in section 2.

3.2 Two-Step Approach for Defining CoCon Type Semantics

This section discusses *how to* formally define the CoCon semantics.

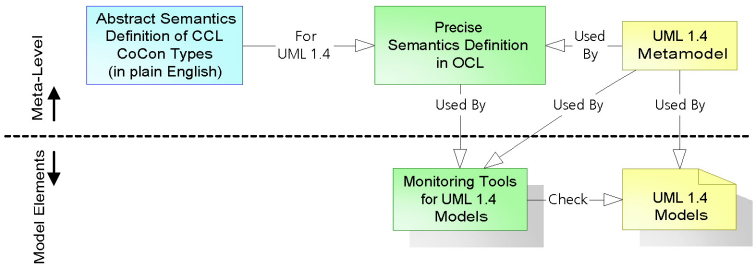

Figure 2. Two-Step Approach for Defining the Semantics of CoCons

CoCons can be applied to artefacts at different development levels, e.g. models at design level or component instances at runtime. Different artefact types or different versions of the same artefact type can be used at each development level. For instance, UML 1.4 models or other specification techniques can be used at the design level. Figure 2 illustrates the two-step approach for defining semantics of CoCon types for the artefact type 'UML 1.4 models':

- The **artefact-type-independent semantics definition of a CoCon type** does not refer to specific properties of an individual artefact type as discussed in section 3.3.
- The **artefact-type-specific semantics definition** of a CoCon type refers to a metamodel of a specific artefact type in order to define the semantics in terms and constructs of this metamodel in a formal or semi-formal language. For example, which concepts express `ACCESSIBLE TO` in UML 1.4 models? Section 6 will discuss which UML model elements must (not) be associated in which way with which other UML model elements in order to comply with `ACCESSIBLE TO` CoCons.

3.3 Formalization of Context-Based Constraints

Instead, CoCons are a limited version of predicate logic as described next. A CoCon expresses a condition on how its constrained elements must be related to each other. This condition is called CoCon-predicate. Each element of the target set must relate to each element of the scope set as defined by the polyadic CoCon-predicate. The CoCon semantics can be expressed via the following predicate logic formula:

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

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

Each CoCon-predicate $C(x, y)$ can be combined with the CoCon-predicate operation `NOT` or `ONLY`:

- `NOT` negates the relation $C(x, y)$ as follows: $\forall x, y : T(x) \wedge S(y) \rightarrow \neg C(x, y)$
- `ONLY` is mapped to two propositions:
  - $\forall x, y : T(x) \wedge \neg S(y) \rightarrow \neg C(x, y)$
  - $\forall x, y : T(x) \wedge S(y) \rightarrow C(x, y)$

3.4 Research Related to Context-Based Constraints

The key concept of CoCons is the indirect selection of constrained elements. Any unknown element becomes involved in a context-based constraint simply by having the matching context property value(s). Hence, the constrained 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 element is automatically constrained by a CoCon due to the element's context property values.

A recent and interesting access control policy approach can indirecly select the constrained network elements: the Ponder language for specifying management and security policies is defined in [9]. Three differences between CoCons and Ponder exist. First, CoCons don't have operational semantics, while Ponder has. Second, Ponder does not address distribution requirements, while CoCons do. And finally, Ponder is based on the domain concept discussed in section 2.3. Similar to context conditions, Ponder uses domain scope expression ([24]) for selecting elements according to their domain. A domain, however, consists of a single name, while context properties consist of a name and values.

4 Distribution CoCons

4.1 The Notion of Distribution CoCons

Distribution CoCons determine whether the target set elements have to be available at the CoCon's scope elements or not. 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 'computers' in the scope sets of distribution CoCons are discussed here.

4.2 Distribution CoCon Types

This section proposes several CoCon types for expressing distribution requirements. Each CoCon type can be combined with the CoCon-predicate operation 'NOT' or 'ONLY' after the keyword MUST. For example, the CoCon type `ALLOCATED TO` can either state that certain elements `MUST BE ALLOCATED TO` other elements, or that they `MUST NOT BE ALLOCATED TO` other elements, or that they `MUST ONLY BE ALLOCATED TO` other elements. The abbreviation '`(NOT | ONLY)`' is used to refer to all three possible CoCon-predicate operations of one CoCon type in the next sections.

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 and can also be realised with some middleware platforms. In this paper, the term 'a component is replicated' means that the component's state is serialized and the resulting data is copied. The following CoCon types handle replication:

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

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.

4.3 Examples for Using Distribution CoCons

The 'availability' requirement introduced in section 1.2 can be written down via CCL as follows:

```
ALL COMPONENTS WHERE 'Workflow' CONTAINS 'Create Report' MUST BE
ALLOCATED TO ALL COMPUTERS WHERE 'Operational Area' CONTAINS 'Controlling'
```

.

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

4.4 Related Research on Distribution and Network Policies

One way in which we cope with large and complex systems is to abstract away some of the detail, considering them at an architectural level as composition of interacting objects. To this end, the variously termed *Coordination, Configuration and Architectural Description Languages* facilitate description, comprehension and reasoning at that level, providing a clean separations of concerns and facilitating reuse. According to [13], in the search to provide sufficient detail for reasoning, analysis or construction, many approaches are in danger of obscuring the essential structural aspect of the architecture, thereby losing the benefit of abstraction. On the contrary, CoCons stay on an abstract level in order to keep it simple.

Aspect-oriented languages supplement programming languages with properties that address design decisions. According to [12], these properties are called *aspects* and are incorporated into the source code. Most aspect-oriented languages do not deal with expressing design decisions in during *design*. D^2AL ([3]) differs from the other aspect oriented languages in that it is based on the system model, not on its implementation. Objects that interact heavily must be located together. D^2AL groups collaborating objects that are directly linked via associations. It describes in textual language in which manner these objects

interact which are connected via these associations. This does not work for objects that are not directly linked like 'all objects needed in the 'Create Report' workflow.

Darwin (or 'δarwin') is a *configuration language* for distributed systems described in [16] that, likewise, expresses the architecture explicit by specifying the associations between objects. 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 invoke each other or not. Distribution CoCons allocate objects together because of shared context instead of direct collaboration. They define a context-specific cluster. Distribution CoCons assist in grouping related objects into **subject-specific clusters** and define how to allocate or replicate the whole cluster.

Recent work by both researchers ([7]) and practitioners ([18]) has investigated how to model non-functional requirements and to express them in a form that is measurable or testable. Non-functional requirements (also known as quality requirements) are generally more difficult to express in a measurable way, making them more difficult to analyse. They are also known as the 'ilities' and have defied a clear characterisation for decades. In particular, they tend to be properties of a system as a whole, and hence cannot be verified for individual system elements. The distribution CoCon types introducesd here specify non-functional requirements. Via the two-step semantics definition, these CoCon types can clearly express 'ilities'. They are particulary helpful in expressing crosscutting ilities that apply to more than one system element, because one CoCon can select several involved elements according to their context property values.

5 Detectable Conflicts of Distribution CoCons

If distribution requirements CoCons contradict each other then an tool interpreting the CoCons will not be able to perform an action appropriately because one CoCon negates the effect of the other. Thus, it is important to have a means of detecting and resolving any conflicts that arise. A conflict between constraints arises if they express opposite conditions on the same elements. This section defines **inter-CoCon conflict detection constraints** (short: **inter-CoCon constraints**). First, three general inter-CoCon constraints are presented that apply to each of the distribution CoCon types defined in section 4.2. In predicate logic, each of these CoCon types can be expressed as $C(x, y)$. The first inter-CoCon constraint can apply if one CoCon has a `NOT` operation, while another CoCon of the same CoCon type hasn't.

General Inter-CoCon Constraint 1: The two CoCons $\forall x, y : T_1(x) \wedge S_1(y) \rightarrow C(x, y)$ and $\forall x, y : T_2(x) \wedge S_2(y) \rightarrow \neg C(x, y)$ contradict each other if $\exists x, y : T_1(x) \wedge T_2(x) \wedge S_1(y) \wedge S_2(y)$.

If $C(x, y)$ is defined as `x MUST BE ALLOCATED TO y` CoCon-predicate then this general inter-CoCon constraint states that no element x must both be

`ALLOCATED TO` and `NOT ALLOCATED TO` any y. For instance, the following privacy policy informing forbids to manage personal data on web servers: `ALL COMPONENTS WHERE 'Handles Personal Data' CONTAINS 'True' MUST NOT BE ALLOCATED TO ALL COMPUTERS WHERE 'Installed Software' CONTAINS 'Web Server'`. According to the general inter-CoCon constraint, this privacy policy contradicts the availability-policy presented in section 1.2 if a web server is installed on any computer used by the controlling department.

The next two inter-CoCon constraints take the CoCon type operation `ONLY` into account. The semantics of the operation `ONLY` are defined in section 3.3. The following inter-CoCon constraint applies if one CoCon without CoCon type operation contradicts another CoCon with the CoCon type operation `ONLY`:

General Inter-CoCon Constraint 2: The two CoCons
- $\forall x, y : T_1(x) \wedge S_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : T_2(x) \wedge \neg S_2(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_2(x) \wedge S_2(y) \rightarrow C(x, y)$

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

The following inter-CoCon constraint applies if one CoCon with the CoCon type operation `ONLY` contradicts another CoCon with the CoCon type operation `ONLY`:

General Inter-CoCon Constraint 3: The two CoCons
- $\forall x, y : T_1(x) \wedge \neg S_1(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_1(x) \wedge S_1(y) \rightarrow C(x, y)$ and
- $\forall x, y : T_2(x) \wedge \neg S_2(y) \rightarrow \neg C(x, y)$
  $\forall x, y : T_2(x) \wedge S_2(y) \rightarrow C(x, y)$

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

Besides the general inter-CoCon conflict detection constraints, *CoCon-type specific* inter-CoCon conflicts exist as listed next. The elements e_i and e_k are target or scope set elements of distribution CoCons with $i \neq k$. An inter-CoCon conflict exists if any of the following inter-CoCon constraints is violated:

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

Inter-CoCon conflicts can be handled by assigning a different priority to each CoCon – 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. This section has demonstrated that conflicting CoCons automatically can be detected by checking the constrained elements (identified via context conditions) if they violate one of the inter-CoCon constraints defined above. Hence, the conflicting distribution requirements can automatically be detected. This is a major benefit of CoCons.

6 The CoCon Type Semantics for UML

Software tools can support software designers in monitoring the artefacts of a development process for compliance with distribution CoCons if the artefact-specific semantics of the distribution CoCon types are defined. This paper focuses on checking UML models for compliance with distribution CoCons. Hence, this section defines the artefact-type-specific semantics of the `ALLOCATED TO` CoCon type for UML 1.4 models via OCL.

In UML, deployment diagrams show the configuration of software components. Software component instances represent run-time manifestations of software code units. A deployment diagram is a graph of nodes connected by communication associations. An arrow (= dependency) or the nesting of a component symbol within a node symbol maps specified the deployment of a component at a node. As illustrated in the UML deployment diagram shown in figure 3, the component type 'ContractManagement' is deployed on the computer type 'Laptop'. According to the CoCon in section 4.3, 'ContractManagement' must not be allocated to the 'Laptop' because this computer belongs to the controlling department.

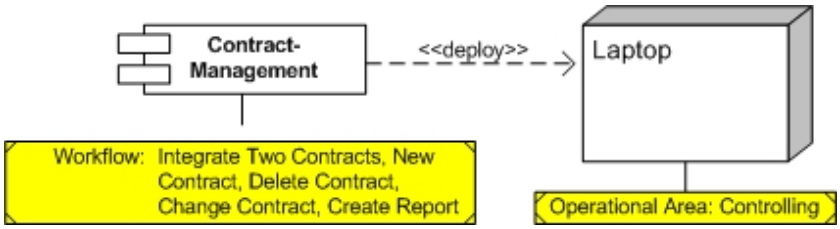

Figure 3. A Deployment Diagram Showing a Component Type that violates the Availability Requirement of Section 4.3

The `ALLOCATED TO` CoCon given in section 4.3 can be translated into the two following OCL expressions. The first OCL expression referts to component *types* that are allocated to nodes via deployment associations:

```
context node inv:
        self.taggedvalue -> select(tv | tv.dataValue =
        "Controlling")
        .type -> select(td | td.name = "Operational Area")
        -> notEmpty()
implies self.deployedComponent
        -> select(c | c.oclIsTypeOf(component))
        self.taggedvalue->select(tv | tv.dataValue = "Create
        Report")
        .type -> select(td | td.name = "Workflow")
        -> notEmpty()
```

This OCL statement states that a nodes having the tagged value 'Operational Area: Controlling' must (= `NotEmpty()` in OCL) contain those components having the tagged value 'Workflow: Create Report'. This OCL expression only addresses component types and node types. Nevertheless, distribution also applies to component *instances* and node *instances*. The following OCL expression maps the availability policy to component instances that reside on node instances:.

```
context nodeinstance inv:
        self.taggedvalue -> select(tv | tv.dataValue =
        "Controlling")
        .type -> select(td | td.name = "Operational Area")
        -> notEmpty()
implies self.resident
        -> select(c | c.oclIsTypeOf(componentinstance))
        self.taggedvalue->select(tv | tv.dataValue = "Create
        Report")
        .type -> select(td | td.name = "Workflow")
        -> notEmpty()
```

In case of `ALLOCATED TO` CoCons, the artefact-type-*independent* semantics definition consists of five words: 'x must be allocated to y'. On the contrary, the corresponding artefact-type-specific OCL listing is about much longer because it considers a lot more details. CCL stays on the artefact-type-independent, abstract level. OCL, however, is too close to programming for expressing requirements at this abstraction level. The effort of writing down a requirement in the minutest details is unsuitable if the details are not important. The designer can ignore many details by expressing the same requirement via CCL instead of OCL. Moreover, it is easier to adapted the short artefact-type-independent CCL expression instead of changing all the OCL expressions if the corresponding requirement changes.

As a proof of concept implementation, 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 distribution CoCons into the Design Critiques ([17]) mechanism of ArgoUML. However, it only pro-

totypically demonstrates how to verify UML models for compliance with distribution CoCons. It needs to be improved before using it in production.

7 Conclusion

This paper presents an approach for declaratively defining distribution requirements: context-based *constraints* (CoCons) define what must (not) be allocated to what according to the current context of components and documents.

7.1 Availability versus Load Balance

This paper focuses on availability. However, availability and load balanceare 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. This paper suggests improving availabilty by grouping related objects into *subject-specific* clusters and allocating or replicating the whole cluster via CoCons to the computer(s) where it must be available.

In order to detect conflicts between availability and load balance early, the system load should be considered already in the model. A system load estimations can either be a result of approximation based on common sense and experience, of a simulation as suggested by [14], or of (maybe prototypical) runtime metrics. Automatically detecting conflicts between system load and distribution CoCons is a topic of future research, though.

7.2 Limitations of Distribution CoCons

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

- If the metadata is extracted newly from its element each time when checking a context condition and if the extraction mechanism works correctly then the metadata always is correct and up-to-date. Moreover, the extraction mechanism ensures that metadata is available at all.
- Contradicting context property values can automatically be prevented via value-binding CoCons as explained in [4].
- Additional Metadata can be automatically derived from already existing metadata via belongs-to relations as explained in [4].

Within one system, only one ontology for metadata should be used. For instance, the workflow 'Create Report' 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 ([4]) or Model Correspondence Assertions ([6]). However, not every vocabulary problem can be solved via engineering techniques. These techniques can reduce the heterogeneity, but they cannot overcome it completely. Hence, the need for a controlled ontology remains the key limitation of CoCons.

7.3 Benefits of Distribution CoCons

Context properties can dynamically group elements. They facilitate handling of overlapping or varying groups of elements that share a context even across different element types or systems. Hence, one distribution requirements definition affecting several unrelated elements that even may not be managed by the same platform can now be expressed via one constraint. Context properties allow *subject-specific clusters* to be concentrated on. For instance, all components belonging to workflow 'X' may form a cluster. Other concepts for considering metadata exist, but none of them writes down constrains that reflect this metadata.

Requirements specification should serve as a document understood by designers, programmers and customers. CoCons can be specified in easily comprehensible, straightforward language. In complex domains, no one architect has all the knowledge needed to control a system. Instead, most complex systems are maintained 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 [8] as a general problem in software development. Even the person who specifies distribution requirements via CoCons does not have to have the complete knowledge of the components and computers involved due to the *indirect* association of CoCons to the components and computers involved. It can be unknown what exactly must (not) be allocated where when writing down the distribution requirements.

Distribution requirements can change often. The key concept for improving the adaptability of distribution requirements definition is indirection — the constrained elements can be indirectly selected according to their metadata. The metadata can be easily adapted whenever the context of a component or computer changes. Furthermore, each deleted, modified or additional component or computer can be automatically considered and any resulting conflicts can automatically be identified. CoCons automatically adapt to changed contexts, components or computers.

References

1. Thomas Baker. A grammar of dublin core. *D-Lib Magazine*, 6(10):47–60, october 2000.
2. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.
3. Ulrich Becker. D^2AL - a design-based distribution aspect language. Technical Report TR-I4-98-07 of the Friedrich-Alexander University Erlangen-Nürnberg, 1998.
4. Felix Bübl. The context-based constraint language CCL for components. Technical Report 2002-20, Technical University Berlin, Germany, available at www.CoCons.org, October 2002.
5. 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.
6. Susanne Busse. Modellkorrespondenzen für die kontinuierliche Entwicklung mediatorbasierter Informationssysteme. PhD Thesis, Technical University Berlin, Germany, Logos Verlag, 2002.
7. Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic, Boston, 2000.
8. 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.
9. Nicodemos Damianou. A policy framework for management of distributed systems. PhD Thesis, Imperial College, London, UK, 2002.
10. Keith Devlin. *Logic and Information*. Cambridge University Press, New York, 1991.
11. Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.
12. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming ECOOP*, volume 1241 of *LNCS*, pages 220–242, Berlin, 1997. Springer.
13. Jeff Kramer and Jeff Magee. Exposing the skeleton in the coordination closet. In *Coordination 97, Berlin*, pages 18–31, 1997.
14. 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.
15. OMG. UML specification v1.4, September 2001.
16. Matthias Radestock and Susan Eisenbach. Semantics of a higher-order coordination language. In *Coordination 96*, 1996.
17. 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.
18. Suzanne Robertson and James Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.

19. Edward Sciore, Michael Siegel, and Arnon Rosenthal. Context interchange using meta-attributes. In *Proc. of the 1st International Conference on Information and Knowledge Management*, pages 377–386, 1992.
20. Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using semantic values to falilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems (TODS)*, 19(2):254–290, 1994.
21. Amit P. Sheth and Sunit K. Gala. Attribute relationships: An impediment in automating schema integration. In *Proc. of the Workshop on Heterogeneous Database Systems (Chicago, Ill., USA)*, December 1989.
22. Morris Sloman and Kevin P. Twidle. Domains: A framework for structuring management policy. In Morris Sloman, editor, *Chapter 16 in Network and Distributed Systems Management*, pages 433–453, 1994.
23. Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, Reading, 1997.
24. Nicholas Yialelis. Domain-based security for distributed object systems. PhD Thesis, Imperial College, London, UK, 1996.