Towards Designing Distributed Systems With ConDIL

Felix Bübl

Technische Universität Berlin, Germany
Computergestützte InformationsSysteme (CIS)
{ fbuebl@cs.tu-berlin.de }
http://cis.cs.tu-berlin.de/~fbuebl

Abstract Designing and maintaining a distributed system requires consideration of dependencies and invariants in the system's model. This paper suggests expressing distribution decisions in the system model based on the system's context. Hence, UML is enriched by two new specification techniques for planning distribution: On the one hand, *'Context properties'* describe dependencies on the design level between otherwise possibly unrelated model elements, which share the same context. On the other hand, *'context-based distribution instructions'* specify distribution decisions based on context properties. The distribution language 'ConDIL' combines both techniques. It consists of four layers introduced informally via examples taken from a case study.

Keywords: Designing Distributed Systems, System Evolution, Design Rationale, Resource Management

Published: in Proceedings of 2^{nd} EDO in Davis, California in November 2000 on page 61–79 of LNCS Nr. 1999 © Springer Verlag

Renamed: In all later publications the 'context-based distribution *instructions*' are called 'context-based distribution *constraints*', and 'ConDIL', is called 'DCL' (Distribution Constraint Lanuage).

1 Introduction

1.1 Distribution Needs 'Design for Change'

The context for which a software system was designed continuously changes throughout its lifetime. *Continuous software engineering* is a paradigm discussed in [11,17] and in KONTENG[1] to keep track of ongoing changes and to adapt legacy systems to altered requirements. The system's design level must support these changes - it must be prepared for changes. It must be possible to safely transform the system model in *consistent modification steps* from one state of evolution to the next without unwanted violation of existing dependencies and invariants.

[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

This paper suggests taking basic distribution decisions into account right from start of the software development process and expressing these decisions in a distribution language on the design level.

1.2 Distribution needs 'Design for Implementation'

An increasing number of services and data have to be spread across several distributed sites due to modern business requirements. Arranging the distribution of services and data is a crucial 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. According to [7], this approach is inefficient and 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. Section 2 discusses typical design errors, which can be avoided, if essential distribution aspects are already considered in analysis and design.

1.3 Case Study: Government Assisted Housing

The results presented here are based on a case study in cooperation of the Senate of Berlin, debis and the Technical University of Berlin. In this case study, distribution across offices in 23 districts of a complex software system for housing allowance administration was designed.

2 Problem: Distribution Needs to be Considered From the Start

The main problem addressed here is how to reach overall performance and availability while distributing a complex software system in a way that eases later modification. Typical design problem are discussed here. First section 2.1 examines the need to consider distribution during the analysis phase. Afterwards distribution in the design phase is observed in section 2.2. Then evolution in distributed systems is addressed in section 2.3, before the goals of this paper are summarized in section 2.4.

2.1 Preparing Distribution in the Analysis Phase

Already throughout the analysis phase key aspects, which determine the distribution of a software system, should be addressed. The process of developing a software system starts with a *requirements engineering* stage where functional and non-functional requirements of the system are identified. Functional requirements determine a system's functionality. Non-functional requirements describe the quality that is expected from the system. Two typically aspects ignored so far will be now analysed. On the one hand, existing infrastructures and resources –

hardware, software or organisational – might determine distribution and, therefore, must be reflected in analysis phase. On the other hand, a rough prediction of the most frequently used services and data and their usage should be given:

1. Which workflows are most commonly used and how often is each of them executed?
2. Which objects are heavily used and what quantity of each of them will have to be managed by the system?
3. Along which basic contexts shall the system be cut? What are the key factors that determine the allocation of services or data onto a node?

Application area experts can often answer these questions easily. They can estimate these numbers or name the most frequently occurring workflows already in the analysis phase. In most cases, this important information is not investigated sufficiently. Without considering a given infrastructure and the crucial system load it cannot be determined whether a software system is well balanced, scalable or reliable later on. The abovementioned analysis results need to be reflected in the subsequent design phase.

2.2 Planning Distribution in the Design Phase

Up to now, a distribution decision has typically been taken in the implementation phase. But the rationale for distribution decisions may be ignored later on if they are already not expressed in the model. For instance, by writing down the distribution requirements '*all data needed by the field service must be stored on the field worker's laptops*' at the design level, the developers will not be allowed to remove certain data from the laptops in future modifications. One distribution requirement may contradict others. The field service example may contradict a requirement stating that '*personal data must not be stored on laptops*'. In order to reveal conflicting distribution requirements and to detect problems early they should be written down in the beginning of the development process, not during implementation. Fixing them during implementation is much more expensive.

While defining the *structure* of a system in the design phase the assignment of an attribute to a class[2] can complicate or even inhibit distribution.

Each class is used in several *contexts*. For instance, a system's context is a company with both headquarters and field service. The notion of 'context' is explained in section 3.1. Up to now the context of a class has not been considered in the design phase. Thus, it cannot be determined during design whether one class contains attributes from different contexts or not. If, e.g., a class belonging to the context 'headquarters' also contains attributes of the context 'field service' the instances of this class have to be available in both contexts and therefore be replicated or remotely invoked. Unnecessary network load can be avoided during design if this class is split into classes that only contain attributes of one context.

[2] In the case of designing a non object-oriented system please substitute 'class' with 'entity-type' throughout this paper

Multimedia data or other large binary data may obstruct replication. For instance, the class '*Movie*' has a large attribute '*Moviefile*' that stores a large MPEG movie. And it contains additional normal attributes like '*Movietitle*', '*Producer*' and '*Last time when this movie was accessed*'. If this class is replicated, high network traffic occurs each time when the '*Last time when this movie was accessed*' attribute changes. This attribute is modified each time the movie is watched, and each time the whole changed instance of this class including the many megabytes large '*Moviefile*' must be copied over the network. This error could be recognised already on the design level, if replication would be considered during the design phase.

The *dynamic* behaviour of a system must also be reflected in distribution decisions. But dynamic models can become highly complex. This paper suggests taking only the names of the most often used workflows into account for planning distribution. These names can be investigated in the analysis phase and play an important role here. Up to now a system model doesn't show, which classes are needed by which workflow or database transaction. Designers should attempt to allocate all the classes needed by one workflow onto the same node to avoid network or system overload. There is no technique yet available to assist distribution decisions according to essential workflows or database transactions.

One distribution requirement can apply to several model elements, which can be part of different views. It should take dependencies between model elements into account. Current specification techniques for dependencies do not allow for model elements which are not directly connected or related, or are not even part of the same specification or view.

Considering distribution on the design level allows for the early prediction of problems and facilitates modifying the system model as discussed in the next section.

2.3 Modifying a Distributed System

Today *many* people develop *many* parts in *many* languages of *many* views of **one** distributed software system. This leads to system models that contain huge numbers of different model elements, like classes in class diagrams or transitions in petri nets. It gets increasingly difficult to understand such complex 'wallpaper'. Besides other problems, two important tasks become hard to fulfil:

Distribution Decisions: In deciding upon the allocation or replication of one model element, other distribution decisions and dependencies between related model elements must be considered.

System Evolution: The modification of one model element should take existing dependencies and distribution instructions into account in order not to violate them, and it should be propagated to all other concerned model elements. As analysed in [14], existing design techniques typically concentrate only on forward systems management.

New techniques for denoting dependencies and distribution instructions are needed to reduce the high costs of rearranging a distributed system model.

2.4 Goal: To Plan Distribution on the Design Level

This paper suggests enriching a UML model by two new specification techniques that facilitate decisions about a system's distribution and support consistent modification steps:

'Context properties' describe dependencies on the design level between otherwise possibly unrelated model elements.

'Context-based distribution instructions' are invariants on the design level. They specify distribution requirements for sets of model elements that share a context. Thus, they facilitate the preservation of distribution constraints at runtime or during model modifications.

The distribution language 'ConDIL' combines both techniques to express essential distribution requirements. It consists of four layers introduced informally via examples taken from a case study. Before introducing ConDIL in section 4, the following section presents the new techniques in general.

3 Context-Based Instructions

The essence of 'ConDIL' is writing down key distribution constraints. Before explaining why the Object Constraint Language OCL is not used, the two other techniques applied instead are sketched here. Context properties are initially discussed in section 3.1. They establish a basis for the context-based instructions introduced in section 3.2.

3.1 Describing Indirect Dependencies via Context Properties

System models contain elements, like classes in class diagrams or transitions in petri net diagrams. Model elements can depend on one another. Modification of a system model must not violate dependencies between model elements, and distribution decisions should take these dependencies into account. In order to consider dependencies, they must be specified in the model. Model elements can relate to each other even if an association does not directly link them. A new technique for describing such correspondences was introduced in [3]: Context properties allow the specification of dependencies between otherwise unrelated model elements that share the same context – even across different views or specifications.

A graphical representation and informal definition is indicated in figure 1. The context property symbol resembles the UML symbol for comments, because both describe the model element they are attached to. The context property symbol is assigned to one model element and contains the names and values of all the context properties specified for this model element: the context property named *'Workflow'* in figure 1 has the value *'merging two contracts'* for the class *'Contract'*. Thus, it connects a static class to dynamic workflow that may be specified elsewhere - e.g. in a petri net.

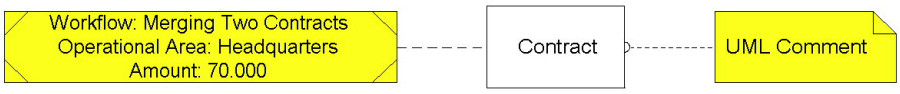

Figure 1. Enhancing UML via Context Properties.

A context property has a name and a set of possible values. Both are investigated in the analysis phase. Due to the limited length of this paper, methodical guidance is not discussed. The examples in the following sections distinguish between functional and non-functional context properties and give hints on how to use them. The 'context properties' used in this paper on design level differ from the 'context properties' used in Microsoft's COM/.NET on implementation level. A future paper will deal with context properties on the implementation level, but this paper focuses on the design level only.

Context properties are a technique that allows handling the results of analysis phase in the subsequent design phase. This general purpose grouping mechanism leads to better-documented system models and improves their overall comprehensibility. Knowing background information about elements enhances understanding of the model. It allows to focus on distributing or modifying within *subject-specific, problem-oriented views*. For example, only those model elements belonging to the workflow *'merging two contracts'* are of interest in a distribution or modification decision. Knowing about a shared context is necessary in order not to violate existing correspondences while distributing or modifying a model.

The primary benefit of enriching model elements with context properties is revealed in the next section, where they are used to specify a new type of invariants.

3.2 Introducing Context-Based Instructions

In the previous section context properties were introduced as technique for describing dependencies between otherwise unrelated model elements. This section suggests expressing decisions based on these correspondences as '*instructions*' which are invariants on the design level and specify requirements. There are many different kinds of *implementation requirements* during different phases of developing a software system. This paper focuses on distribution decisions. Before describing a language for distribution instructions in section 4, two examples are discussed here to promote their general benefits.

In first example, the system's context is a company with both headquarters and field service. Then, according to this context, there is a distribution instruction saying, that certain components have to run on the travelling salesman's laptop without being in connecting to the server. This distribution instruction is

valid for all model elements, whose context property 'operational area' has the value 'field service' – it is a *context-based instruction*. This example is one answer to the question 'along which basic contexts shall the system be cut?' raised in section 2.1 by cutting the system into operational areas. In the case of a context change, like an alteration in the company's privacy or security policy the system must be adapted to the new requirements without violating the already existing distribution instruction. In this case, either no component necessary for working offline should be removed from the field worker's laptop for security or privacy reasons, or the distribution instruction must be adapted. Therefore, modifying the model must take already existing distribution instructions into account in order not to violate them.

Another example demonstrates how ensure distribution requirements via context-based distribution instructions. In order to develop a smoothly operating distributed system every workflow[3] should be able to run locally on a single node without generating network traffic. A frequently executed workflow should be described in a context property 'workflow' with the value 'merging two contracts' to all model elements needed by this workflow, and than state a distribution instruction allocating everything needed by this workflow onto the same node.

Classical load balance calculations need non-functional context properties instead of functional ones like 'workflow'. Some of the examples in section 4 illustrate how to use context-based instructions in predicting and establishing an evenly load balance in distributed systems.

Fundamental choices about how to distribute a model should reflect the system's context and should be preserved and considered in a modification step. Therefore they must be expressed on the design level. Up to now, there has not been a method or technique for describing reasons for distribution decisions. A language for specifying distribution requirements is proposed in the next section.

4 The Distribution Language 'ConDIL'

The **con**text-based **D**istribution **I**nstructions **L**anguage 'ConDIL' consists of four layers or views enhancing UML:

The enhanced UML class diagram identifies the classes to be distributed.
The net topology layer indicates the hardware resources available in the distributed system.
The distribution instructions layer states distribution decisions for sets of classes of the enhanced class diagram.
The enhanced, generated UML deployment diagram displays the results of the other layers' specifications.

The layers are introduced informally via simplified examples taken from the case study. Before describing each layer the following section addresses limitations of ConDIL.

[3] In the case of designing a distributed database system please substitute 'workflow' with 'transaction' throughout this paper

4.1 Designing distributed Components or Databases with ConDIL?

The short descriptions of the four layers given above speak about distributing 'elements' of the enhanced class diagram. It depends on the type of distributed system to choose a more concrete term: In the case of a *component-based system* 'components' 'classes' or 'objects' are distributed, while in designing a *database*, it is intended to distribute 'entity-types' or 'entities'. The general version of ConDIL proposed here neither fits all the needs of component-based systems nor of distributed databases.

Neither case is discussed in this paper due to space limitations. Forthcoming papers will study each case separately. This paper restricts itself to allocating 'classes' on 'nodes' without going into detail about whether this means allocating a class in a component's interface or in a local schema of a distributed database system. Among the other unexplored topics of interest also are, e.g., heterogeneity, the choice of distribution technologies or the allocation of model element other than classes.

The general version of the distribution language ConDIL allows the specification of any kind of distributed system because it only expresses the most basic distribution requirements necessary.

4.2 ConDIL's Enhanced UML Class Diagram

The previous section explained why ConDIL has been restricted to allocating classes to nodes. Classes are specified in an enhanced UML class diagram. The example illustrated in figure 2 describes a system where poor people can apply for a state grant that helps them to pay their rent. The housing allowance is an n:m association between the applicant and the apartment and lasts only for a certain time. The following concepts have been added to the UML Class Diagram:

Association Qualifiers & Classes are standard UML techniques which are used here to specify *foreign keys* for each association. Up to now foreign keys were not specified in the conceptual design. This well-known relational database concept is necessary to enable cutting an association when the classes at its ends are allocated onto different nodes. The concept of foreign key attributes for 1:n or n:m associations is explained in standard database literature. As illustrated between 'Apartment' and 'Owner', the arrow at 1:n-associations is required to give their qualifiers a clear semantic. An n:m association like between the classes 'Applicant' and 'Apartment' needs an *association class* – 'Housing Allowance' – to hold two foreign key attributes: in the qualifier of the association class for both foreign keys the name of the class referred to is followed by the name of the foreign key attribute.

Context Properties were introduced in section 3.1. Each context property has a name, e.g., 'Operational Area' and values, e.g., 'Headquarters'. A legend shows all the valid values for each context property. The default value of each context property is underlined. In the case the context property has only the

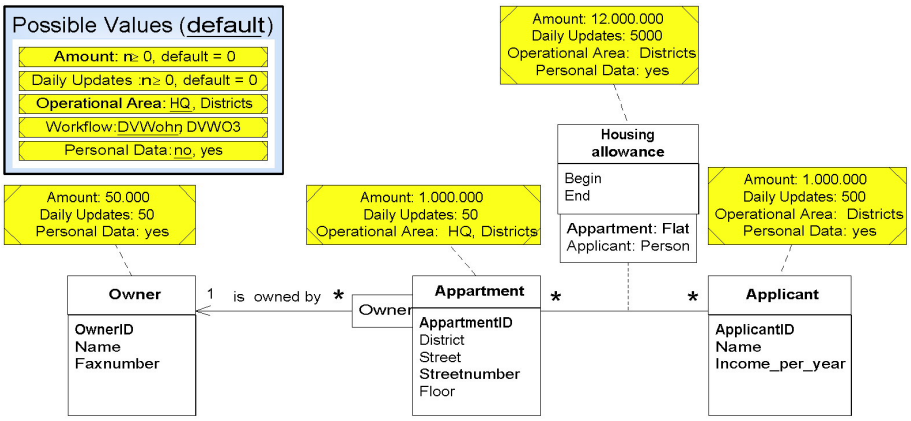

Figure 2. Extract of an Enhanced UML Class Diagram Showing the Case Study.

default value for a class, it can be left out in the context property symbol of this class. In this example, the context property 'Workflow' has the default value 'DVWohn'. As all classes in figure 2 belong to this workflow, none of them needs to specify it in its context property symbol. In the next section, figure 3 shows an alternative approach, where – for improved comprehensibility – the designer spelled out the values of all context properties for each class even if it is the default value.

The context properties suggested here may be useful for planning a distributed system in general. When designing a particular system some of these proposed context properties may be ignored and some unmentioned ones may be added by the developer, as needed. Figure 2 exemplifies both functional and non-functional properties:

'Workflow' describes the functionality – it is a context property of *functional type*. It reflects the most frequent workflows or use cases and enables the designer to write down distribution requirements for these workflows. For instance, in a distributed system all of the model elements needed by a certain workflow should be allocated to the same computer in order to be able to execute this workflow without connecting to the network. This requirement can be verified by marking all of the concerned model elements accordingly. Thus, *static aspects of system behaviour* can be expressed.

'Operational Area' is another example for managing distribution via a functional context property. It enables the writing down of the distribution decisions for certain departments or domains. Functional context properties provide an organisational perspective and thereby facilitate software design as indicated by [12].

'Personal Data' signals when a class contains data that must not be distributed due to its intimate content. It is of functional type and exemplifies how to model roles or authorization via context properties.

'Amount' is of *non-functional (qualitative) type*. Its value holds the estimated number of instances of each class.

'Daily Updates' allows to distinguish between frequently changing and rather static data. This non-functional information is needed to estimate the network load later on.

Hiding avoidable dynamics by only considering *static aspects of behaviour* is the primary benefit of using functional context properties. They allow consideration of methods, services, sequence or operations and others details to be left out in distribution decisions. Otherwise the model complexity would increase rapidly. The goal of this paper is to keep distribution decisions as straightforward as possible.

Section 4.5 give a demonstration how to use qualitative (non-functional) context properties for calculating metrics in order to facilitate system assessment.

After having only slightly extended the standard UML diagram up to now, two totally new visual languages are proposed in the next two sections.

4.3 ConDIL's Net Topology Layer

Taking distribution decisions demands awareness of hardware resources. The net topology layer depicts the hardware resources. It identifies nodes and connections as demonstrated in figure 3:

A node is denoted by a symbol resembling a computer. By working with symbolic names like 'RAID-Server' the same net topology diagram can be deployed for different customers and cases, and it does not have to be changed if the hardware is replaced. Symbolic names can describe hardware requirements or services like 'database', 'sensor', 'router' or 'printer'. A constraint stated for *one* symbolic node, e.g. 'Laptop', applies to *all* nodes of this kind. By using symbolic names, the network topology diagram must not be modified if the number of laptops changes.

A Connection links nodes and is drawn with the standard UML deployment diagram symbol for connections.

Both nodes and connections are more closely described via context properties. In the net topology diagram mostly 'non-functional' context properties are used. They are needed later on for establishing reasonable load balance. For instance, the non-functional context property 'Speed' can be helpful in taking a distribution decision. Improving a system's load balance via ConDIL is discussed in section 4.5.

The net topology diagram will feature additional symbols like 'table space' for databases or 'container'for Enterprise Java Beans. These specific symbols will be published in articles concentrating on certain platforms.

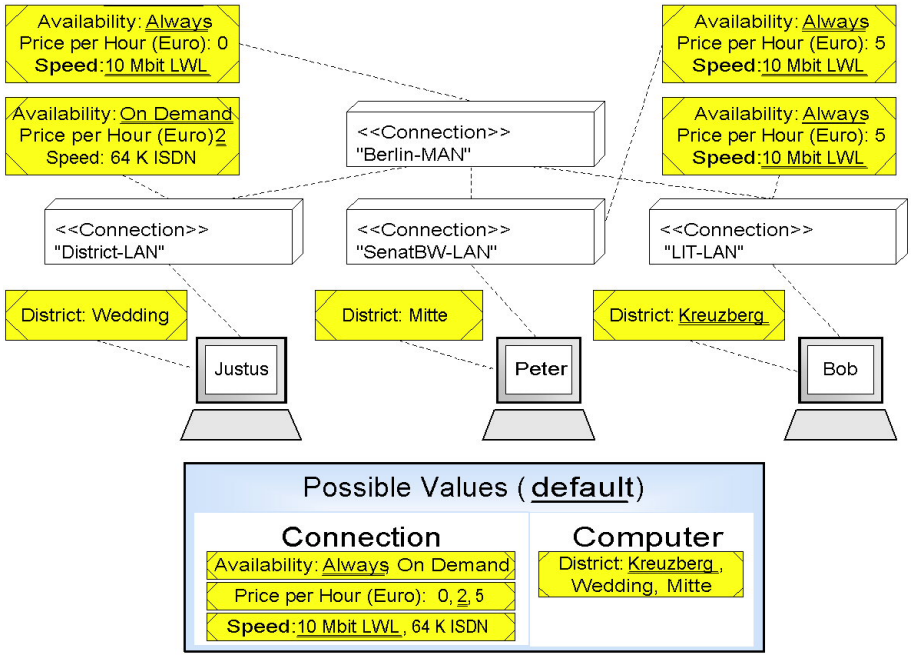

Figure 3. The ConDIL Net Topology Layer.

It would be possible to include distribution instructions in the net topology diagram, but there are already several symbols associated with one node, and there will be even more in future enhancements of this layer. Distribution instructions are better depicted in their own layer, which is introduced now.

4.4 ConDIL's Distribution Instructions Layer

In this central ConDIL layer two different types of distribution instructions control either *allocation* or *replication* decisions. As shown in the next figures, a distribution instruction is drawn as follows: an arrow points from the *symbol specifying the elements involved* that shall be distributed to the *symbol specifying the target* where the elements involved shall be allocated. The allocation instructions are introduced first.

Context-Based Allocation Instructions specify allocation constraints for more than one target. The basic assumption for discriminating between allocation and replication instructions is that there is *exactly one master copy* of each instance of a class. In the case of assigning one class to several nodes, only one node holds an original instance. All other copies of this instance on other

nodes are replicated from this master copy. This notion enables the underlying middleware or database to ensure consistency.

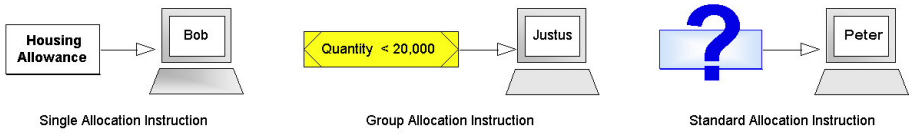

Figure 4. Allocation Instructions of the ConDIL Distribution Instructions Layer.

You shouldn't confuse classes with instances in this section. The general version of ConDIL introduced in this paper describes distribution on the class level only. It enforces all original instances of one class to be allocated on the same node. The upcoming version of *ConDIL for databases* will stick to the class level as well. In contrast, the future version of *ConDIL for components* will deal with allocation on instance level and will allow original instances of the same class on several computers.

In order to assure the allocation of a class on only one node different priorities are given to each kind of allocation instructions. They are exemplified in figure 4 from left to right:

Single allocation instructions are demonstrated on the left side of figure 4. They have the highest priority – no opposing set allocation instruction applies for a class, if a single allocation instruction for it exists. It is not allowed to spell out contradicting single allocation instructions for the same class.

Set allocation instructions are illustrated in the middle part of figure 4. They apply for every class that fits to the *context condition* and is not allocated via a single allocation instruction. In the example given, all classes having the context property 'Quantity' with values of less than 20,000 are allocated onto the node 'Justus'. Being able to give set instructions is the main benefit of ConDIL. Grouping classes, which share the same context, allows taking distribution decisions based on essential requirements. If a class is assigned to different nodes via contradicting set allocation instruction, one assignment becomes the master copy, and all others become synchronous replicates of this master copy.

One default allocation instruction as shown on the right in figure 4 must be specified in a system model. It has the lowest priority and applies only to classes where no other allocation instructions apply.

Context-Based Replication Instructions are exemplified in figure 5. The symbols are explained now from left to right:

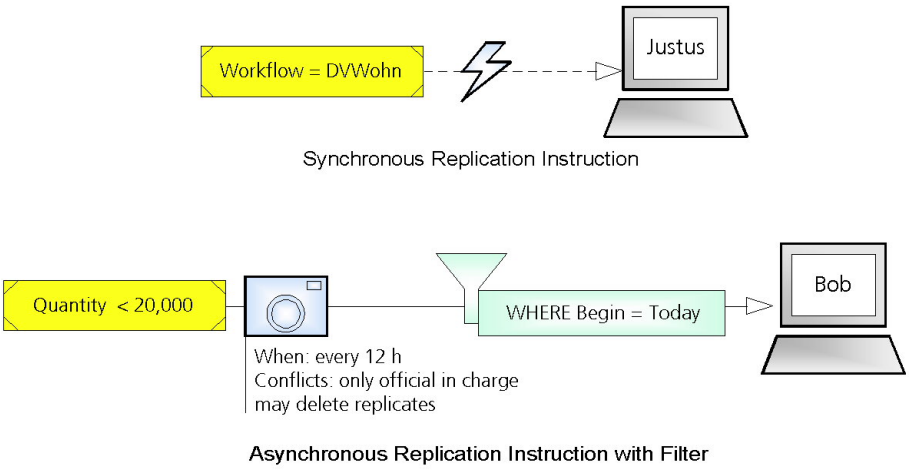

Figure 5. Replication Instructions of the ConDIL Distribution Instructions Layer.

A camera represents asynchronous replication. It needs additional attributes that are placed directly under the camera symbol. The attribute 'When' must be indicated in any case because asynchronous replication cannot be implemented without knowing when to replicate. The attribute 'Conflicts' must be stated only in the case of writeable or deleteable replication and describes what to do in the case of update or deletion conflicts. This information may be omitted if default rules are given for all asynchronous replication instructions when to replicate or what to do in the case of conflicts.

The dotted arrow of a replication instruction indicates *write only* replication.

A lightning bolt represents synchronous replication as demonstrated on the top of figure 5.

Set instructions , again, are the most powerful application of ConDIL. Available database products possess fast group replication mechanisms as Oracle's 'refresh groups' for the implementation of set replication instructions. Contrary to allocation instructions, there are no different priorities for replication instructions, and there is no default replication instruction.

The Filter symbol is needed to express views, where only some of the instances of the source class(es) shall be replicated. Normally filters only make sense for single replication instructions, because they use an SQL WHERE clause

that names attribute of the selected class(es). In a filtered set instruction, like in figure 5, all classes selected by the context condition 'Quantity less then 20,000' must have the attribute 'Begin'. This could happen in historical data warehouses, but in general not all classes selected by the context condition have the attribute(s) needed by the filter condition.

Overall, this section introduces the key concepts in an informal way. Both the details of the context-based distribution language ConDIL and adequate methodical guidance are topics of future research. The next section suggests automatically generating an enhanced UML deployment diagram showing the results of the requirement specifications.

4.5 ConDIL's Enhanced, Generated UML Deployment Diagram

Without illustrating the results of a specification stated in the other three layers, a designer cannot verify if the goals of designing a distributed system have been reached: reliability, scalability and load balance need to be confirmed in an overview diagram showing all the details. ConDIL uses an enhanced UML deployment diagram for validation of the distribution goals and for *architectural reasoning* as discussed in [15]. The enhanced deployment diagram can automatically be generated based on the specifications in the other three layers.

Figure 6 shows one possible result of ConDIL distribution design. In this example, not all of the goals have been accomplished yet. The classes have been successfully allocated and replicated in a way that the workflows or database transaction on 'Justus' and 'Bob' can be executed locally. Thus, both computers can continue working even if the network fails. But in ensuring reliability, the other aims (scalability and load balance) are not achieved. In the example, both the connection 'District-LAN' and the computer 'Bob' show overload.

The numbers representing the system load in figure 6 are derived from the non-functional context properties given in the other layers. They can either be automatically calculated via given formulas, or they can be intellectually estimated via common sense. Some standard context properties have been suggested, but standard formulas for load balance calculation are a topic of future research. At the EDO symposium three precision levels for optimising load balance were discussed:

1. Estimation based on common sense and experience is the quickest and most vague way to figure out load balance.
2. Simulating an UML model as proposed by [9] turns out better load numbers, but needs an arbitrary detailed modelling of the system behaviour.
3. The best but most expensive load prediction results from prototypical benchmarking.

The enhanced deployment diagram assists in detecting problems before implementation phase. Figure 6 demonstrates the need to change the other ConDIL layers until a satisfactory trade off between the contradicting goals of distribution design is reached.

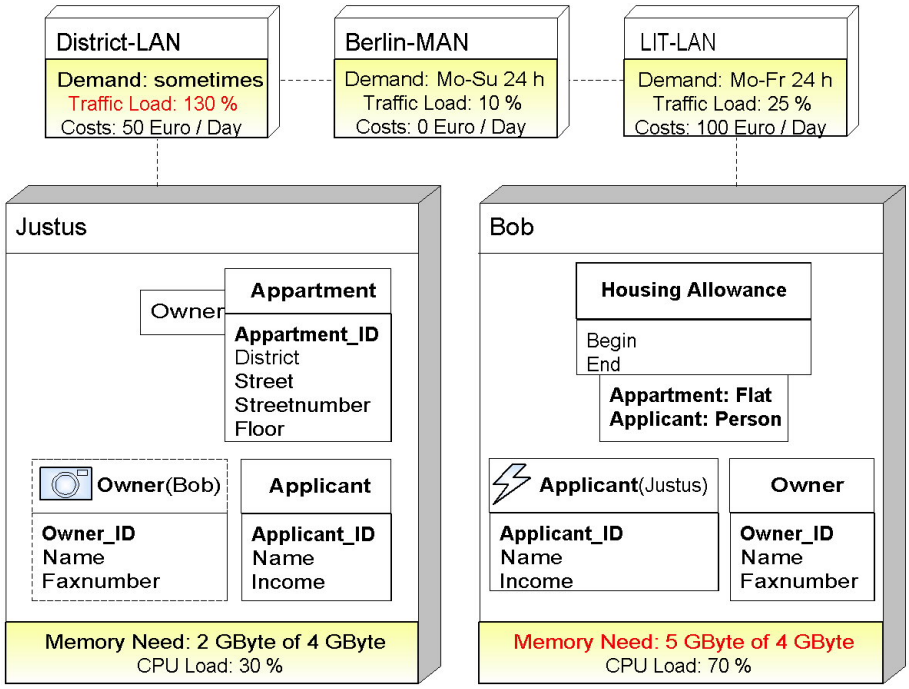

Figure 6. Displaying a System Overview with ConDIL's Generated Enhanced UML Deployment Diagram.

5 Related Research

5.1 Planning distribution

After distributed applications became popular and sophisticated in the 80is, developers needed techniques to support their development. According to [1] over 100 new *programming languages* specifically for *implementing* distributed applications were invented. But hardly anyone took distribution into consideration already on the *design* level, whereas ConDIL does. Research concentrated on dealing with parallelism, communication, and partial failures on implementation level. On the contrary, ConDIL describes distribution on the design level.

For instance, the *Orthogonal Distribution Language (ODL)* proposed in [5] supplements the programming language C++ for distributed processing and, thus, does not support distribution decisions during the design phase. Roughly the same applies to *aspect oriented languages*, which, in principle, resemble the idea of context properties. D^2AL as one of these aspect oriented languages, however, differs in that it is based on the system model, not on its implementation. Though the closest to ConDIL, the differences will now be discussed.

According to [2], D^2AL enables the programmer to specify the distribution configuration based on UML. Whereas ConDIL states a distribution instruction for a group of classes which share the same context, D^2AL groups collaborating objects that are directly linked via associations. Objects that heavily interact must be located together. D^2AL describes in textual language in which manner those objects interact which are connected via associations. This does not work for objects that are not directly linked like 'all objects needed by the field service'. In contrast, ConDIL is a visual approach based on a shared context instead of object collaboration.

For companies, staying competitive means meeting continuously changing business requirements. As presented in [18], especially business process dominated systems demand flexible models. This paper proposes enriching system models via distribution decisions, which can be based on important business processes. In this paper the context property 'Workflow' was used to increase the adaptability of a model by enriching it via invariants according to its business processes.

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 components. 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 [8], 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. ConDIL is a concise and simple language explicitly designed for describing architectural distribution structures.

Darwin (or 'δarwin') is a 'configuration language' for distributed systems described in [13] that makes architecture explicit by specifying the connections between software agents. Instead of describing explicitly connections between distributed objects, ConDIL expresses vague dependencies via contexts properties and writes down instructions for model elements that share a context.

In [12] *policy driven management for distributed systems* integrates organisational engineering and distributed databases. Replication models are described within the organisation model. This way, consistency of replication policies can be inferred from the organisational model. In ConDIL, distribution instructions write down replication requirements. ConDIL can assist policy driven management for distributed systems.

There used to be a lot of interest in machine-processed records of *design rationale*. According to the many authors of [10] the idea was that designers would not only record the results of their design thinking using computer-based tools, but also the process that they followed while designing. Thus, the software designer would also record a justification for why it was as it was and what other alternatives were considered and rejected. Context-based instructions are one way to record design decisions. The problem is that designers simply don't like to do this. The challenge is to make the effort of recording rationale worthwhile

and not too onerous for the designer. In return for writing down design decisions via context-based instructions they harvest several benefits sketched below.

5.2 ConDIL and UML

In the long term ConDIL proposes becoming part of UML. This section inspects each new ConDIL concept, whether it can be realised via standard UML extension mechanisms or not:

Context properties can either be based on UML tagged values or on UML comments. Using tagged values is the concept closest representing context properties, because both are key-value pairs. Typically one class is characterized by several context properties. Each tagged value floats around its class separated from the other tagged values belonging to the class. This can be confusing when numerous tagged values belong to the same model element. Therefore, a specialisation of the UML comment symbol is proposed in this paper to place all of the context properties and their values of one class *into one single symbol*.

No stereotypes or packages: In order to write down distribution instructions for *sets of classes* it is necessary to *group* classes that share a context. ConDIL describes the context of classes via the new technique 'context properties' rather than grouping classes of the same context via existing UML mechanisms. Usually several context properties exists with each of them having multiple values. In [3] mechanisms, like *UML stereotypes* or *UML packages*, are rejected, because they cannot handle several overlapping contexts. Usually quite a few context properties exist where each has multiple values. For instance, if your system has n different values for 'Workflow', you would need $2^n - 1$ different stereotypes to classify your classes. Considering an additional context property, e.g. 'Operational Area' would result in an even more confusing number of stereotypes.

The enhanced class layer of ConDIL can be implemented via standard UML extension mechanisms.

The net topology layer is not part of the UML at present and calls for enhancements of the UML metamodel.

The context-based distribution instruction layer cannot be derived from contemporary UML as well. Context-based instructions are constraints associated with context properties (tagged values). The *Object Constraint Language OCL* summarized in [16] is the UML standard for specifying constraints like invariants, pre- or post-conditions and other constraints in object-oriented models. OCL was not chosen for stating context-based distribution instructions for several reasons. The model should serve as a document understood by designers, programmers and customers and, therefore, should use such simplified specification techniques. ConDIL is an easily comprehendible, straightforward visual language separated into layers and reduced to the bare minimum of what's needed for designing distribution. A major distinction is that *one* OCL constraints refers to *one* model element,

while *one* context-based instructions refers to *many* model elements. Even the OCL 1.4 draft does not permit one constraint for several model elements.

The enhanced deployment diagram can be realized via standard UML extension mechanisms. UML deployment diagrams are not widely considered to be attractive – e.g. [6] describes them as 'informal comic'. ConDIL improves their expressiveness and demonstrates their reasonable usage in developing distributed systems.

6 Conclusion

6.1 Limitations of ConDIL

Up to now ConDIL is only used on design level. An upcoming paper will examine the influence of ConDIL on the handling of distributed objects at runtime. ConDIL is restricted to *static aspects of system behaviour*. Reducing behaviour to the names of frequent workflows ignores a lot of the information that may have an impact on distribution decisions. For instance, the ConDIL layers do not show if and how workflows depend on each other. But such detailed modelling of dynamics would hardly improve the load balance calculation or the verification of the other design goals. The quality of the load balance prediction wouldn't fairly increase by accurate consideration of behaviour, because all non-functional numbers are estimated anyway and won't turn out a precise prediction. On the contrary, ignoring dynamical aspects provides some of the benefits that are summarized in the next section.

6.2 Benefits of ConDIL

ConDIL supports the design of distributed systems from the start of the development process. Key distribution requirements can now be expressed at the design level. It detects problems already during design. And it eases the identification of essential dependencies and invariants and thus improves the readability of the model, facilitates distribution decisions and helps to prevent their violation in later modification steps.

ConDIL's enhanced deployment diagram assists in establishing a trade off between load balance, reliability and scalability because it provides for the consideration of relevant aspects, such as names of frequent workflows or existing hardware resources. Detailed modelling of dynamical system behaviour is not needed for assessment of distribution decisions. Fewer iterations in the development process are necessary when distribution requirements were already taken into account during design.

When one model element changes, other related elements might also have to be adapted. Maintenance is a key issue in *continuous software engineering*. ConDIL helps ensure consistency in system evolution. A ConDIL instruction serves as an invariant and thus prevents violation of distribution requirements in a modification step. It helps detect when design or context modifications compromise intended functionality. The dependencies and invariants expressed via

ConDIL can prevent unanticipated side-effects during redesign, and they support collaborative distributed design. It is changing contexts which drive evolution. ConDIL's instructions are context-based and are therefore easily adapted to context modifications.

6.3 ConDIL Roadmap

Currently a CASE tool capable of ConDIL concepts is implemented at the Technical University of Berlin by extending the tool 'Rational Rose'. A first prototype is available for download at `http://cis.cs.tu-berlin.de/~fbuebl/ConDIL`.

The following subjects will be addressed in future research:

Enhancing ConDIL: Two versions of ConDIL will be published for designing either distributed component-based systems or distributed databases.

Method: Proposing a new technique is pointless without developing an adequate method for its appropriate use. For instance, both context properties and context-based instructions could be acquired during the reengineering process. Recovering basic dependencies and invariants can outline the distribution architecture of a legacy system. Furthermore, guidelines will be developed how to combine ConDIL and ConCOIL ([4]). – the **Con**text-based **C**omponent **O**utline **I**nstruction **L**anguage introduced in [4]. It describes the logical architecture of a component-based system.

Algorithm & tool for consistent modification: An algorithm for maintaining consistency in a modification step by considering existing context-based instructions will be developed. In a first step, this algorithm will serve to generate the enhanced deployment diagram. As 'proof of concept' a tool for evolution support will be implemented. This tool won't be based on the already existing Rational Rose extension, because ConDIL turned out to overstrain Rose's extensibility.

ConDIL at runtime: An upcoming paper will examine whether a scheduler can exploit ConDIL constraints for the sake of an optimal load balance.

References

1. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.
2. Ulrich Becker. D^2AL - a design-based distribution aspect language. Technical Report TR-I4-98-07 of the Friedrich-Alexander University Erlangen-Nürnberg, 1998.
3. 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.
4. Felix Bübl. Towards the early outlining of component-based systems with ConCOIL. In *ICSSEA 2000, Paris*, December 2000.
5. M. Fäustle. An orthogonal distribution language for uniform object-oriented languages. In A. Bode and H. Wedekind, editors, *Parallel Comp. Architectures: Theory, Hardware, Software and Appl.*, LNCS, Berlin, 1993. Springer.

6. Martin Fowler and Kendall Scott. *UML Distilled*. Object Technologies. Addison-Wesley, Reading, second edition, 1999.
7. Peter Koletzke and Paul Dorsey. *Oracle Designer Handbook*. Osborne/McGraw-Hill for Oracle Press, Berkeley, second edition, 1999.
8. Jeff Kramer and Jeff Magee. Exposing the skeleton in the coordination closet. In *Coordination 97, Berlin*, pages 18–31, 1997.
9. 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 Volker Gruhn, Wolfgang Emmerich, and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, LNCS, Berlin, 2000. Springer.
10. Thomas P. Moran and John M. Carroll, editors. *Design Rationale : Concepts, Techniques, and Use (Computers, Cognition, and Work)*. Lawrence Erlbaum Associates, Inc., 1996.
11. Hausi Müller and Herber Weber, editors. *Continuous Engineering of Industrial Scale Software Systems*, Dagstuhl Seminar #98092, Report No. 203, IBFI, Schloss Dagstuhl, March 2-6 1998.
12. Antonio Rito Silva, Helena Galhardas, Paulo Sousa, Jorge Silva, and Pedro Sousa. Designing distributed databases from an organisational perspective. In *4th European Conference on Information Systems*, Lisbon, Portugal, 1996.
13. D. Spinellis. δarwin reference manual. Technical report, Dept. of Computing, Imperial College, London, 1994.
14. Stefan Tai. *Constructing Distributed Component Architectures in Continuous Software Engineering*. Wissenschaft & Technik Verlag, Berlin, Germany, 1999.
15. Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio. Architectural reflection realising software architectures via reflective activities. In Volker Gruhn, Wolfgang Emmerich, and Stefan Tai, editors, *Engineering Distributed Objects (EDO 2000)*, LNCS, Berlin, 2000. Springer.
16. Jos B. Warmer and Anneke G. Kleppe. *Object Constraint Language – Precise modeling with UML*. Addison-Wesley, Reading, 1999.
17. Herbert Weber. IT Infrastrukturen 2005 - Informations- und Kommunikations-Infrastrukturen als evolutionäre Systeme. White Paper, Fraunhofer ISST, 1999.
18. Herbert Weber, Asuman Sünbül, and Julia Padberg. Evolutionary development of business process centered architectures using component technologies. In *Society for Design and Process Science, IEEE International Conference on Systems Integration IDPT*, 2000.