Max Mühlhäuser, Wolfgang Gerteis, and Lutz Heuser

DOCASE: A METHODIC APPROACH TO DISTRIBUTED PROGRAMMING

Distributed object-oriented programming languages have proved to be superior to other known approaches for the development of complex distributed applications. Most of these languages support location-independent method invocation and object migration. Together with the fine granularity of objects, these features allow *distribution-transparent* modeling and programming. However, large object-oriented applications tend to exhibit huge numbers of objects and intertwine several operational software aspects for which a more methodic approach is required. The DOCASE project (*D*istribution and *O*bjects in *CASE*) was established to identify fundamental elements of such a method approach.

There is an increasing demand for complex multiparty distributed applications, such as cooperative office workflow systems. A predominant problem in the development of these applications is the request for distribution abstraction, which comprises "natural" distribution aspects such as locality, remote invocation, or availability. In the past, process-oriented or RPC-based approaches led to client/server (C/S) computing. Thereby, the boundaries between client and server processes had to be determined early in the software life cycle. However, solutions to many distribution problems have to go beyond C/S computing to meet their requirements [17].

Provided that some basic design rules are observed, the object-oriented approach leads to distributed applications consisting of large numbers of "small" objects. Such "fine-grained" applications can be distributed over the target distributed system at installation time (e.g., with the goal to balance processor load or to minimize internode communication). Thereby, local and remote method calls are absolutely equivalent on the syntactic level. In order to determine the (initial) mapping of objects to network nodes at installation time, distributed object-oriented systems usually offer special language elements [12] or even dedicated "configuration languages" [15].

Moreover, most distributed object-oriented programming languages, such as Emerald [2], DOWL [1], Guide [5], and COOL++ [13], allow *migration* of objects at run time (i.e., sending them to another node). Many such languages allow migration of objects even during their invocation.

To summarize, distributed object-oriented programming languages support distribution-transparent programming extremely well. Additionally, they exhibit the known advantages of the object-oriented approach in general, such as data encapsulation, fine-grained modularity, reusability, and extendability.

There are, however, major difficulties in using current distributed object-oriented technology. The DOCASE project tried to address these. Three particular problems and the solutions developed in DOCASE are discussed in this article: common representation of design artifacts and their implementation; modularization of algorithms by separating dynamic distribution aspects from application algorithms; and guidance throughout the design process. We will present the *hybrid language* DODL, the *superimposition* language T^sL, and DOCASE *design assistant* with its design method description language. These three approaches together provide a unique level of design and implementation support for complex distributed applications.

They have a language-based approach in common since we are convinced that language support is crucial to provide extensive tool support.

Accompanying Example

To guide the user through the various concepts of DOCASE we provide a sample distributed application. The problem to be solved is an office application that allows processing of travel expense claims. The various steps to be supported are: An employee fills in an electronic travel claim sheet; his or her cost center manager signs it; a clerk checks if it conforms to all rules; accounting sends out a payment; and finally the travel claim gets filed. Employees are expected to work on desktops while the cost center manager works on a department server; the clerk who checks the rules and accounting use an office server; and filing is done on a database server.

The application should coordinate the workflow throughout a distributed system (i.e., the enterprise network all the involved employees are working on). All these steps should be affected only by the availability of the node of the actual performer, but not by those used by the other employees.

Hybrid Language

DOCASE attempts to enhance the distributed object-oriented approach with better support for design and structured programming. In particular, the decision was taken to integrate design and implementation into a seamless phase, supported by a single *hybrid language,* called DODL (*DO*CASE *D*esign *L*anguage). The inherent advantage of using a single language is the avoidance of an artificial barrier between the two phases which would mark a virtual "end of design" and "beginning of implementation" and would make it hard to feed implementation-level changes back to design. A single language can, in addition to this, support attempts to shorten the software life cycle, to introduce concurrent engineering, and to apply iterative rapid prototyping or spiral software development. The risk, of course, is that neither of the two phases will be supported sufficiently or that clearly consecutive and distinct steps of software development will be mangled.

A particular strength of DODL lies in its dual graphical and textual notation: each language element has, in addition to its textual syntax, a graphical depiction associated with it. DODL is a design language due to its graphics aspects, structuring support, support for incomplete and iteratively refined design, and animation of incomplete designs. DODL is also an implementation language in that it includes a full-fledged computational model, a broad range of statements, all elements of other distributed object-oriented languages, and a complete configuration language.

Rather than present the language in-depth, we will concentrate on its key concept, namely, *object categories,* and explain them by using their graphical notations.

DODL Object Categories

The credo of the object-oriented community, "Everything is an object," has become common knowledge. This phrase means that "the object" is the rich yet singular syntactic and semantic element of object-oriented languages. It is rich because of its many facets such as data abstraction, typing, inheritance, and message-based method invocation. It is singular since "traditional" distinctions like "functions and data," "client and server," "active and passive," "information entity and information-processing entity," "procedure call, information flow, and control flow" vanish in the object-oriented context.

Instead, the software engineers are responsible for introducing such distinctions, for example, by providing different top-level object types. But then, these distinctions exist mainly in the mental model of the software engineers, exhibited through the *names* given to the different object types. Design and programming tools will *not* be able to distinguish and check different kinds of objects or actions other than by checking signatures (type, object, and method identifiers, call parameters).

DODL *type categories* provide predefined semantics for different notions of application types captured by various language constructs. This means, for instance, that

- Category names are keywords of the language.
- Category-specific characteristics are collected in specific sections of a type description.
- Category-specific actions of objects are captured in special statements of the language.
- Other tools such as the visual design/programming tool [14] are aware of the characteristics of different type categories, too.

Of course, the crux in all this is the "right" choice of type categories, guided by the question: which are the key categories of a "typical" distributed application, as seen by a software engineer? Based on our several years of experience with the development of complex distributed applications [4, 10, 16], and drawing from the experience of many senior software engineers, we have chosen the type categories described later.

In almost all cases, a sketch of a complex (procedural or object-oriented) software system consists of "major" software modules (processes, clients/servers, subsystems), of relations between them (denoting communication, control flow), and of masses of "little objects" (information, events) exchanged between the "major" modules along the relations. This intuitive understanding leads to the three key and most crucial type categories, called *configured type, generated type,* and *relation type.* In addition, locality is the key distribution aspect reflected by the category *logical node.*

Configured Type

Configured types are used to design the basic software architecture of an application, which we call "application configuration." Intuitively, configured objects (i.e., instances of types belonging to the category *configured type*) are "long-living" or "stable"; adding or removing one of them is a major act of "configuration change." While other objects refer to one another via references held in instance variables, configured objects

are explicitly interconnected via special references called *connections*. Thus, configured objects and the connections between them form a skeleton of the application.

In Figure 1 the skeleton of the travel expense application is shown. Each organizational unit is represented by a configured type. The order of execution implies the knowledge about other configured types through *connections*.

Subcategories of configured type. In order to provide for better hierarchical structuring of application configurations, the (semantically poor) object-oriented aggregation concept was enhanced to that of *subsystems;* they denote a set of configured objects, their relations, and management. *Active types* schedule their own threads and may exhibit characteristics of actors or agents if designed accordingly. *Environment agents* cope with the popular legacy problem: they provide object-oriented "wrappers" around existing software packages with which one would typically have to interface from a complex integrated application.

Figure 2 refines the categorization (see Figure 6 for the subcategories). Each employee works concurrently to others, so they are modeled as *active types*. "Accounting" is an organizational unit that represents a group of clerks and encapsulates to whom the action actually gets delegated. Therefore, it is modeled as a *subsystem*.

Generated Type

In contrast to configured objects, *generated objects* are created and manipulated (mainly by configured objects) without affecting the configuration. Intuitively, generated objects are created in masses and may have a rather short lifetime. Mails, routeslips, messages might be examples of these.

The travel claim is a generated type because its instances are generated by a configured-object "initiator" and manipulated by other configured objects (see Figure 3). Furthermore, even a travel expense system without any travel claims (i.e., an empty system), is valid whereas a missing configured object corrupts the execution of the travel expense process.

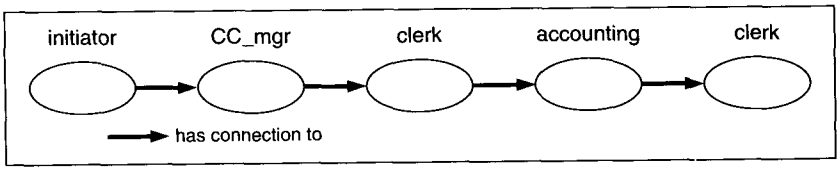


Figure 1. Application skeleton

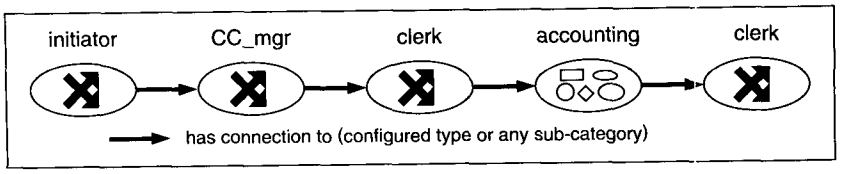


Figure 2. Application skeleton (refined)

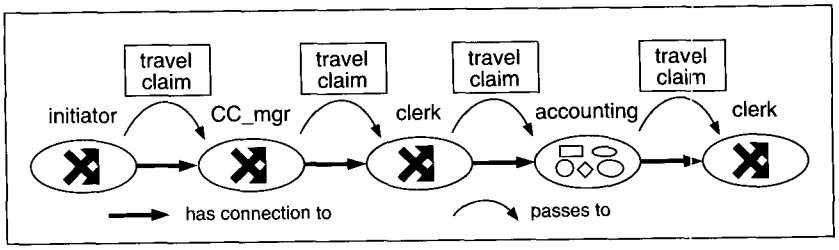


Figure 3. Information flow

Relation Type

Object-oriented languages usually offer nothing but instance variables for modeling all kinds of relationships among objects and nothing but method calls for modeling all kinds of "flows" along these relationships (control flow, synchronization, information flow, activation). Complex relationships such as time-spanning multiparty interaction patterns [20] "vanish" in the program code: there is no central place where the relation is specified and maintained and where its state is kept; there are no means for typing such relationships. In DODL, *relation types* are used to specify complex semantic relationships. Relations are typically established *between configured objects,* and they may specify details of the *exchange of generated objects.*

Subcategories of relation type. Interaction relations are used to model complex multiparty interaction patterns [20]. The *collocation* category is used to relate objects which may occasionally migrate onto a single node during periods of intensive communication [19].

Long-living, global control flows or "workflows" can be modeled using the *cooperation* category, where several configured objects are *related* via a global control flow. A cooperation is used when none of the configured objects can provide the full functionality but each provides part of it as a partner of the cooperation. The global control flow coordinates the partners by defining the order of execution and by providing the common context.

The design of the sample application changes when a cooperation type is introduced (see Figure 4). Instead of peer-to-peer connections, the configured types are enrolled as partners of the cooperation, not knowing who else belongs to it (unless this information should be visible). The travel claim is part of the common context and therefore referenced by the cooperation object. For each travel claim, a global control flow is launched which is encapsulated in an instance of the travel_expense_cooperation.

Logical Node

This category provides the interface to the physical configuration. At design/programming time, the "application configuration" may refer to "logical nodes," see Figure 5. By mapping logical nodes onto real ones

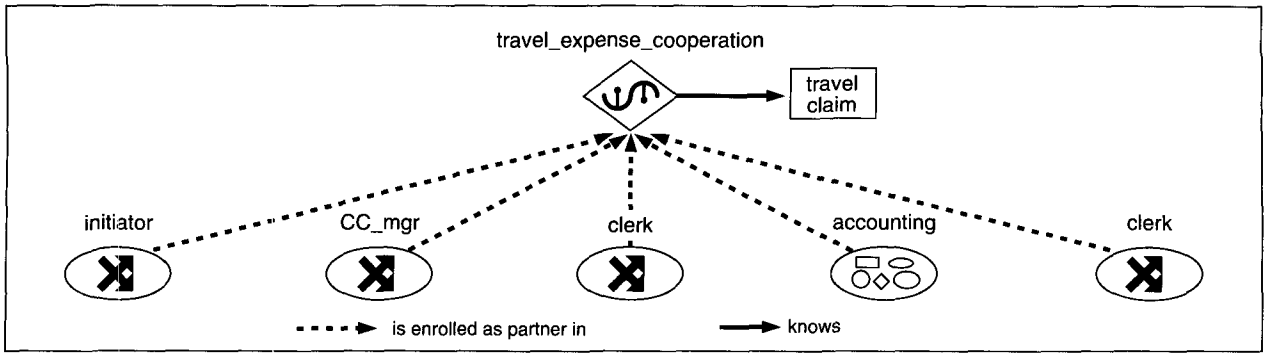


Figure 4. Travel expense cooperation

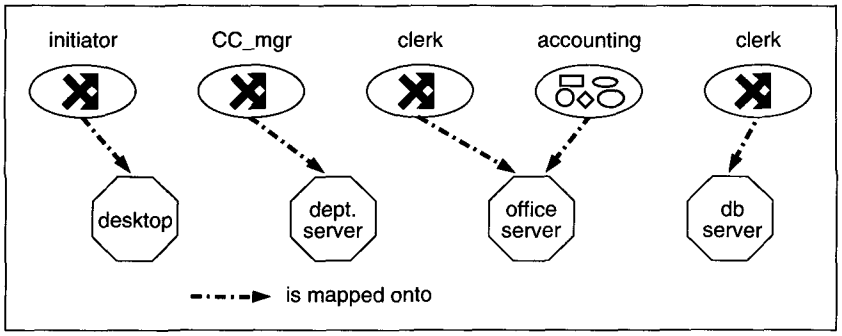


Figure 5. Object placement

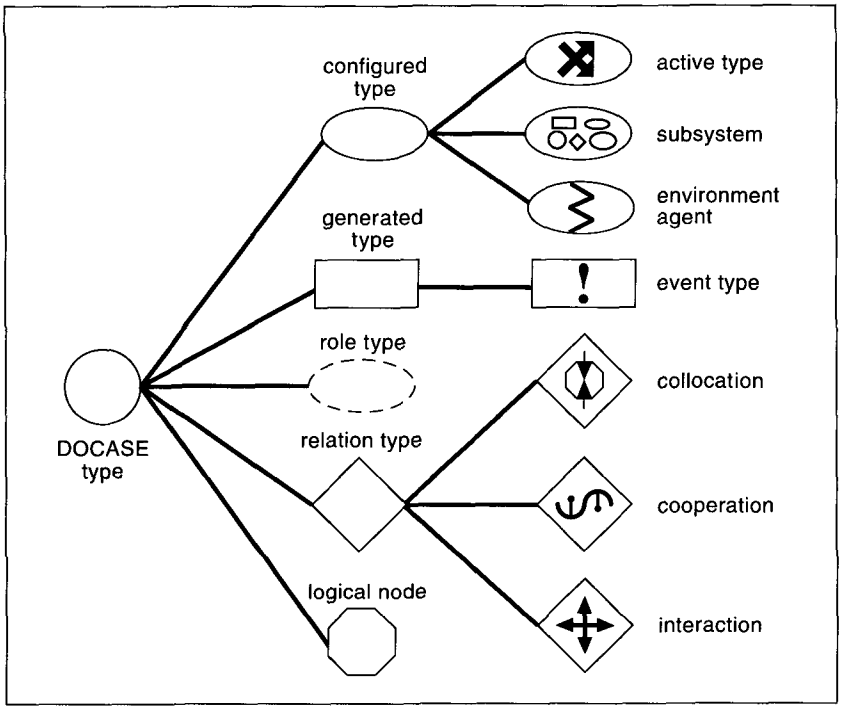


Figure 6. Graphical depiction of DODL categories

at installation time, major aspects of the physical configuration are determined. Logical nodes reflect the granularity of execution (i.e., each one is realized as a single program or workspace executed while the distributed application is running).

For the sake of space, other categories such as *DOCASE type, role type,* and *event type* are not addressed here. For further information we refer to [8] and [9]. Figure 6 shows the icons associated with each of the DODL categories. Graphical design/programming of a DODL application can start by picking, instantiating, customizing, and interrelating such categories in a way similar to the use of WYSIWYG drawing packages.

Concurrency Model of DODL

DOCASE supports a thread-based concurrency model which is divided into three levels, each of which can be developed independently. Threads are treated as components of logical nodes, active types, and cooperations. Synchronization is done at the level of objects, i.e., objects are single threaded. Providing synchronization at a finer granularity, i.e., methods [6, 7], is desirable but beyond the current scope of the *DOCASE* project.

Figure 7 illustrates the concurrency model by excerpting concurrent objects (active objects and cooperations) from the example. Each object is located on exactly one logical node which itself is executed within an operating system (OS) process. Within each OS process, threads are running, each belonging to an object of one of the preceding categories. Parallelism is inherently given by the underlying distributed computer network.

Further Design and Implementation-Level Support

For further design-level support, DODL incorporates, as design aids, several syntactic means to deal with incompleteness of the system under development. The following list of concepts is incorporated into a draft

design of the travel expense cooperation (see Figure 8; keywords are capitalized).

- The keyword TBD may be used instead of a type or variable identifier, allowing for a declarative or structural description of the actual or referenced object; this is particularly important since many object-oriented design approaches determine the instances of an application first and care about types only then.
- For program sections yet to be defined, designers can use the UNDEFINED keyword to make an underspecified DODL program syntactically correct (for the effect see below).
- UNDEF is a valid value for all object references and for simple types and serves as a default value. This helps to trace forgotten initializations, for example.
- UNDEF_BEGIN and UNDEF_END embrace incomplete or unknown parts of an algorithm which are semantically checked as far as possible but for which code cannot be generated.

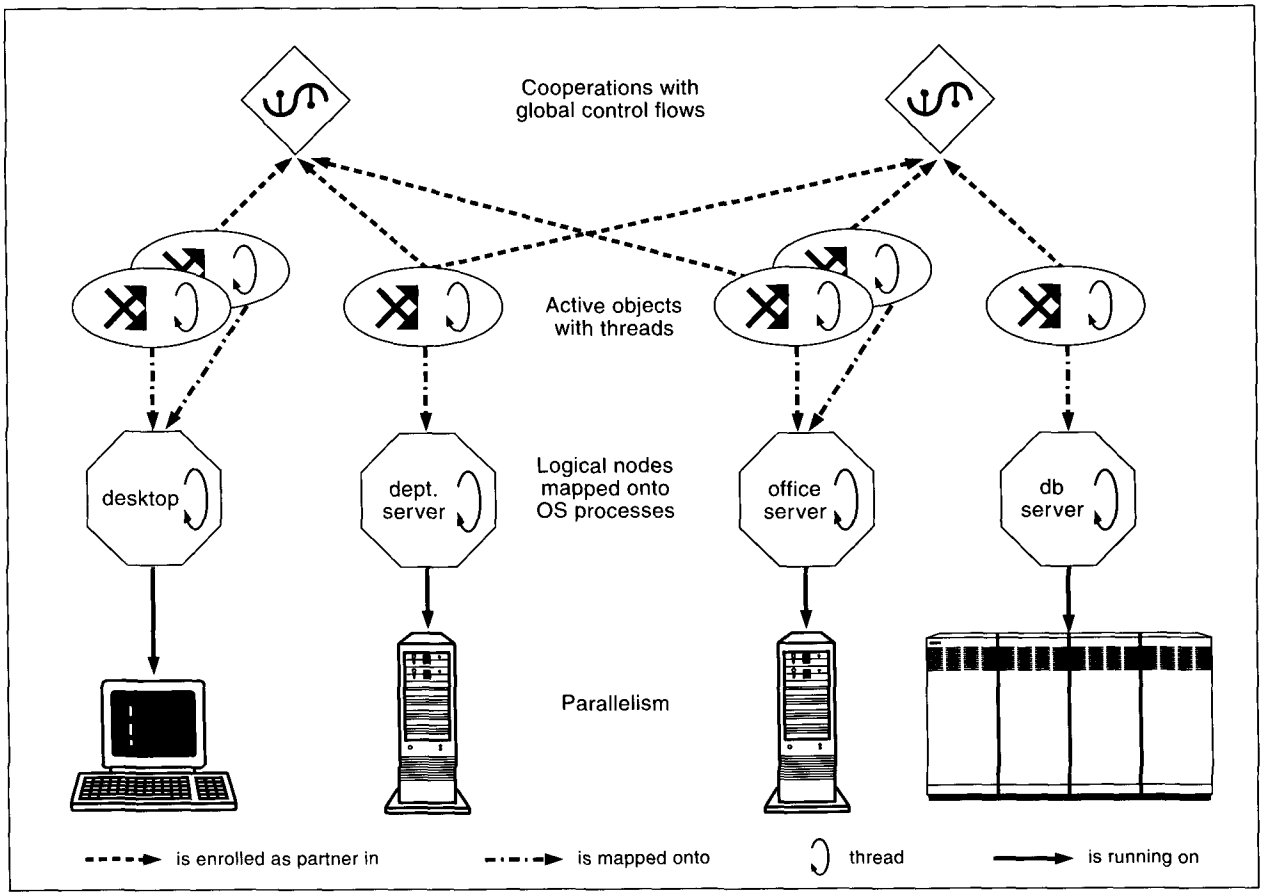
The handling of these constructs is controlled by options of the DODL compiler. It ranges from ignoring them in early design phases, via semi-automatic interactive resolution, to considering all of them as errors in late implementation phases. In particular, the DODL visual programming tool can cope with "underspecified" programs that make heavy use of the preceding statements.

Superimposition

The travel_expense_cooperation is further refined in Figure 9. Each step is executed by an object; coordination is provided by the *global control flow* named exec_tec. We assume that each performing object is located on the node of the person in charge of performing the corresponding step. Taking into account this underlying physical configuration and assuming the cooperation is executed on the workstation of the person initiating the cooperation, the following distribution scenario takes place: The travel_claim object being local to the cooperation results in a large number of remote method invocations during execution, i.e., every time the travel_claim is accessed or the global control flow advances to the next step. This means internode communication is significant. Additionally, if the network is not very reliable these steps risk failure. To overcome these shortcomings, the designer has to increase locality by dynamically migrating the cooperation and its associated travel_claim to the objects performing individual steps.

Considerations like these distribution aspects are called *operational aspects* in DOCASE, since they do not change the functionality of the cooperation. Other such operational as-

Figure 7. DODL concurrency model



```
COOPERATION travel_expense_cooperation
DECLARATIONS
    travel_claim:          TBD ;     ! The type of the travel claim is yet unknown
PARTNERS                  ! Partners are the objects which perform steps of travel_expense
    i:      initiator;
    mgr:    cc_mgr;
    ctr:    clerk;
    a:      accounting;
    t:      clerk;

STATES
    UNDEFINED ;   ! It is not yet defined if a set of pre-defined states has to exist
GLOBAL_CONTROL_FLOW      exec_tec
    BEGIN
        ME.travel_claim :=  UNDEF ;
                          ! Initialization is not yet defined.
        ME.travel_claim := ME.i.fill_in  (UNDEFINED) ;
                          ! The parameter list of 'fill_in' is not yet defined
        UNDEF_BEGIN
                          ! rest of the global control flow is not yet defined
        UNDEF_END
    END
END_COOPERATION
```

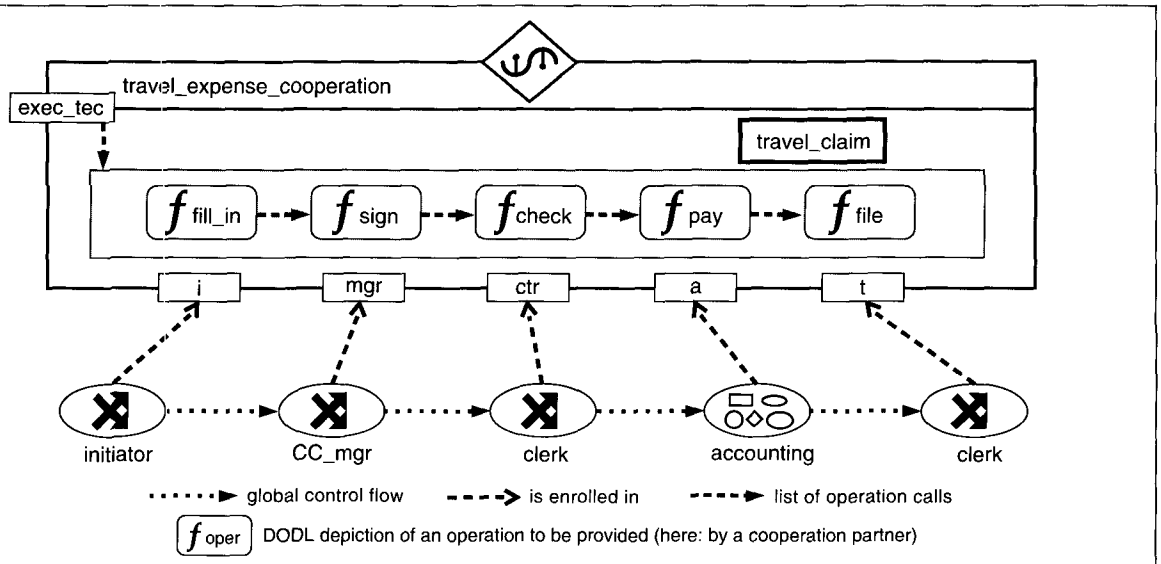


Figure 8. Incompleteness in DODL

Figure 9. Travel expense cooperation: global control flow

pects include security, robustness (e.g., checkpoint/recovery), and accounting.

An easy solution to the preceding problem is the inclusion of *migration statements* before each step in the global control flow. Figure 10 shows the corresponding piece of DODL code. The reader might ask why the run-time system cannot take care of this optimization aspect. Indeed, it could. However, decisions about migration would be made using heuristics based on *a posteriori* knowl-

edge. Therefore, a system would have to gather communication behavior first, which in our example would probably lead to the wrong decision (i.e., migration after being used by the performing objects). Migration statements, however, allow explicit incorporation of *a priori* application knowledge. Obviously, application-oriented distribution aspects at an object level differ from system-oriented distribution aspects at an OS process level, for example, load balancing.

Requirements

In the preceding example, the *operational aspect* interlaces the *basic algorithm*. For larger and more realistic examples this is highly undesirable. Common approaches to structured programming and modularization of software do not provide means to separate algorithms which have to be interlaced at run time. Therefore, even an object-oriented approach to complex applications with several operational aspects will lead to spaghetti code, turning maintenance of an operational aspect (e.g., moving to a new distribution policy) into a nightmare. In order to better maintain, extend, and reuse such operational algorithms, they should be isolated at both design and implementation time.

In DOCASE, we have elaborated the little-known approach of *superimposition* [11] in order to resolve this problem. The principle idea is to layer different algorithms and to allow "higher-layer" algorithms (here: operational ones) to *superimpose* so-called basic algorithms. The major problem stems from the fact that superimposing algorithms depend on the context in which their statements will be executed, making the isolation difficult. This means that for the description of a superimposing algorithm, means have to be provided for "talking" about characteristics of the superimposed basic algorithm.

Apart from modularity, a big advantage can be achieved if superim-

Figure 10. Travel expense cooperation: migration

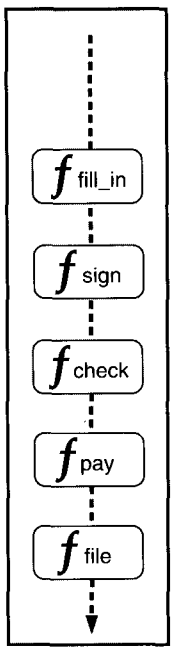
```
COOPERATION    travel_expense_cooperation
DECLARATIONS
     travel_claim:  from_AX4234

PARTNERS            ! Partners are the objects which perform steps of travel_expense
     i:     initiator;
     mgr:   cc_mgr;
     ctr:   clerk;
     a:     accounting;
     t:     clerk;

GLOBAL_CONTROL_FLOW      exec_tec
     BEGIN
          MOVE ME TO ME.i;
          ME.travel_claim := ME.i.fill_in ();
          MOVE ME TO ME.mgr;
          ME.travel_claim := ME.mgr.sign (ME.travel_claim);
          MOVE ME TO ME.ctr;
          ME.travel_claim := ME.ctr.check (ME.travel_claim);
          MOVE ME TO ME.a;
          ME.travel_claim := ME.a.pay (ME.travel_claim);
          MOVE ME TO ME.t;
          ME.travel_claim := ME.t.file (ME.travel_claim);
     END
END_COOPERATION
```

migration statement interlaced with the basic algorithm

step of the cooperation performed by partners



positions can be described in a *generic* way (i.e., that they may be superimposed on *different* basic algorithms). This way, operational aspects can be programmed in a reusable way, drastically reducing software development costs. Support for generic superimpositions requires all identifiers to be in the local scope of the superimposition, described in a way that they can be replaced by actual identifiers of the basic algorithm if necessary.

Superimposition Language

In order to describe superimpositions with the preceding properties, a new language was developed, called T^SL (The Superimposition Language). T^SL requires a host language that provides object-oriented features and a procedural notation of the computational model, as most existing object-oriented languages provide. Thus, T^SL is not limited to DODL but can be seen as a general language architecture.

Interlacing statements. T^SL allows a programmer to describe positions where statements should be implanted. First the interlacing "pieces" of a superimposition are identified, called *implants*. Each of them is associated with a so-called *integration predicate*, consisting of one or more *integration statements*. These contain *selection rules* which identify the positions within the basic algorithm where the implant is to be put. A selection rule consists of two parts: an *execution order identifier* defines the position of the implant relative to the selected statements (BEFORE, AFTER, or CONCURRENT_TO), and a *statement filter* determines all parts of the basic algorithm which have to be superimposed. A statement filter can be either *simple* or *restrictive*. Simple statement filters are *statement classes* such as *assignments, operation call statements,* or *if statements.* The exact number and kinds of statement classes depend on the computational model of the host language.

Restrictive statement filters require further information about the context of statements (i.e., their encompassing statements or internal structure). We identified three groups of restrictive statement filters.

- Statements of a given statement class might be selected only if they contain certain information. For this purpose, *object filters* are offered, described in more detail later.
- Another filter selects statements only if they exist as substatements of control constructs, such as blocks, alternatives, or loops.
- *If statements* should only be selected if their substatements fulfill another statement filter; a third class of restrictive statement filters has to be used (in fact, this class is further subdivided into one for the substatements and one for the statements to be finally selected).

An *object filter* allows selection of statements because of their internal structure. Object filters rely heavily on the capabilities of the host language. The following list of possible object filters is extracted from the current integration with *DODL.*

- Select statements which contain operation calls to certain objects using the object filter AS_CONTROLLING_OBJECT.
- Select statements which call a certain operation (AS_OPERATION).

Further, more specific object filters exist.

Generic integration predicate. In order to reuse superimpositions efficiently, integration predicates have to be as generic as possible. This means, for example, that they have to abstract from the actual identifiers used in the basic algorithms. Therefore, the filters represent a first step toward reusability. However, it is often necessary to express selection rules based on knowledge about the semantics of the basic algorithm, for example, when searching for certain object references or operation names. But even then, an integration predicate should be generic. All names used in an integration predicate should be local to the superimposition and replaced when the superimposition takes place. This can be done with two different kinds of parameters:

- *Replacement parameters* are similar to formal parameters of an operation. They replace all kinds of names with actual ones. At declaration time of the superimposition, actual names have to be given for each replacement parameter. There may be a list of names per replacement parameter, leading to a parallel evaluation of the integration predicates for each name in the list.
- At evaluation time, the first occurrence of a *bind parameter* forces the superimposition tool to bind the current actual name to the bind parameter. For the remainder of the integration predicate, the bound actual name is used for evaluation each time the bind parameter occurs. If a bind parameter binds a list of actual names, the integration predicate is evaluated in parallel using one name per evaluation. The developer of the superimposition does not need to know how often the parameter is bound. Thus bind parameters support genericity and reusability of integration predicates.

Before the parameters can be used, they have to be declared. Since they are to be used for all kinds of names, their correct usage should be checked. For this reason parameters are classified into *operation names, object references,* and *type names.*

Superimposition Example

In this section, we want to exemplify the applicability of T^SL for the integration of operational aspects into arbitrary basic algorithms. For the sake of space, we refer to the migrating global control flow of the *travel expense* example.

Using the superimposition approach provided by T^SL, the cooperation designer can separate the migration statements by using a "migration superimposition," as shown in Figure 11, consisting of a single integration predicate.

The bind parameter obj is necessary, since the superimposition designer would typically not know how many partners of the cooperation exist. Since the parameter is used for object references, it is classified accordingly (OBJECT). The operation filter is as generic as possible, i.e., all operations of the basic type are taken into consideration.

```
SUPERIMPOSITION          migration
    INTEGRATION_PREDICATES
        explicit_migration
            ! declaration of bind parameter
            BIND obj: OBJECT
                ! generic operation filter follows: read as 'all operations'
                IN ANY:
                    ! insert the implant before each identified statement
                    BEFORE
                        ! restrictive statement filter searching for all statements which
                        ! contain an operation call to an instance of a configured type
                            ALL STATEMENTS WHERE
                                AS_CONTROLLING_OBJECT (BIND obj) : CONFIGURED_TYPE
                                    ! implant the migration statement
                                    => MOVE ME TO obj;
END_SUPERIMPOSITION
```

Figure 11. Sample integration predicate for object migration

The selection rule of the integration statement contains the execution order identifier **BEFORE** because the migration of the cooperation object should happen before the next partner will be invoked. The statement filter is restrictive, consisting of a simple filter (**ALL STATEMENTS**) and a **WHERE** clause. Only those statements should be selected which contain invocations to configured types. Thus the object filter has to search for controlling objects which are configured objects. Since the number of partners is unknown, the bind parameter is used to bind the actual names that are needed below. Finally, the implant of the integration statement is given: the migration statement.

Design Assistant

In order to support the design process in the DOCASE environment, a design assistant was developed as a generic tool set. The following sections describe the models on which this tool set is based, concentrating on a language for the description of design methods which is used for customization of the design assistant.

Design Methods

A design method is composed of four major ingredients:

1. *Design elements* are the basic building blocks used during the design process to construct the system under development. They may have a textual and/or graphical notation. Design elements are instantiated into *design artifacts*. These design artifacts are the basic units of the design process. They represent entities of a problem domain or relations among such entities.
2. *Design steps* are operations performed on design elements or design artifacts.
3. A *design procedure* is used to transform the entities of the problem domain into design artifacts of the solution domain. By applying design steps in a correct and reasonable order (as proposed by the design procedure), software quality aspects are assured.
4. A set of *design rules* specifies predefined semantic constraints among design elements and/or artifacts. Note that these rules focus on checking the (maybe intermediate) results of the design process instead of supporting the actual design procedure.

The basic problem of current design techniques lies in the fact that design procedures and rules are given rather informally, as in [3] and [21]. This makes it difficult for software developers to use design steps and elements as intended.

System Architecture

The design assistant consists of three layers: the human interaction layer, the functional layer, and the data layer (see Figure 12). This architecture conforms to the general architectural approach for software development environments as proposed in [17].

The *human interaction layer* consists of a customizable graphical editor. It supports the graphical representation and modification of design artifacts as well as the user interface to the design assistant (for further information see [14]).

The *functional layer* offers a *design method interpreter* together with a design method description language—the heart of the assistant. With this language, a specific design method is specified, which is then interpreted by the method interpreter. The language comprises three parts:

- The *user interface specification* de-

scribes how the functionality required by the design method interpreter is mapped to the facilities offered by the graphical editor.

- The *issue specification* describes the design procedure as a sequence of design issues based on an extension to the issue-based design model as proposed in [18]. The remainder of this section focuses on this central part of the language.
- The *artifact description* provides the specification of different kinds of design elements. As mentioned, both textual and graphical design representation are relevant. Therefore, the current prototype uses an extended parse tree specification as its artifact representation. To obtain the required tree, extensions to compiler generation tools are used (providing for node manipulation, tree traversal, and parse/unparse functionality).

The *data layer* provides an (object-oriented) artifact repository which is not considered here.

Issue Specification

The *issue specification* provides a formal representation of the issue-based design model. It consists of the description of the issues themselves and their composition into a design procedure.

Design issues. A *design issue* is the procedure of solving a design problem by evaluating several alternative solutions and finally making a decision. In an issue, one or more *artifacts* are reviewed: questions are asked about the artifacts; *positions* respond to these questions. Finally, one position is *selected,* and respective design steps are executed.

Arguments *support* or *object* to positions. Two kinds of arguments can be distinguished:

A. *Method-specific* arguments are based on the design elements of a method, i.e., their syntax and semantics. A "good" design method is expected to provide intensive support for these kinds of arguments.

B. *Domain-specific* arguments depend on an application domain and therefore mainly result from the application specification. Providing support for these arguments is rather difficult. It depends on expert knowledge of certain application areas and requires intensive experience about how to use a design method within an application domain.

Without any tool support, the designer has to perform all the tasks himself: having to identify issues raised by previous steps and reflect on the artifacts with respect to these issues. To find an adequate solution for an issue, the designer has to look for alternative positions, select a position, as well as find and appraise arguments. Then the designer has to modify the design by executing the appropriate steps. Most of these activities are today carried out implicitly, without even writing them down for further elaboration. A design assistant can release the designer from these tasks and provide an audited, revisable, and structured path through the design phase.

The issue description is based on the following general considerations:

- As many issues as possible should be predefined and offered to the designer. The "smaller" and "simpler" these issues are, the easier are the steps resulting from the issues. This is important with the automatic execution of steps. Therefore, complex issues should be divided into several subissues.
- The more fixed and unique the sequence of issues is, the better an assistant can help. Exceptions (caused, for example, by errors and forgetfulness of the designer) must be considered.
- Rich argument bases should be offered to the designer, providing both method-specific and domain-specific arguments. Thus the designer's task of appraising arguments can be drastically simplified.
- The steps (belonging to a certain issue) should be executed automatically as far as possible, i.e., the current design status and additional input provided by the designer should automatically lead to a transformation of design artifacts.
- Decision strategies should be specified in order to handle conflicting arguments selected by the designer from the arguments provided.

Design process: Sequences of issues. In realistic design methods, collections of issues are important. However their specification is one of the most difficult aspects of a design method description. Collections of issues are helpful especially for novice designers who need strict guidance within both complex methods and methods that handle design elements in a certain order. The following types of collections are supported by the design assistant:

- *Threads of issues* define a strict sequence of issues for a set of artifacts. Within a design method, there may be multiple independent threads of issues. A thread is finished if the sequence of issues is performed for all artifacts of the set.
- *Check lists* specify a set of closely related issues for a set of artifacts. A check list is finished if all its issues are completed for the set of artifacts.
- A *thread of artifact refinements* is derived from the issues specified for a certain design method, the current state of an artifact, and its refinement history. Support for this kind of collection is still in a very experimental stage and needs more investigation in general.

An example issue. As a sample we present the issue of identifying class candidates as proposed by the object-oriented method "Responsibility-Driven Design" [21]. According to the method, first all nouns are extracted from a textual problem specification. From the list of nouns class candidates are identified. Figure 13 shows this formulated as an issue for our assistant prototype, realized as a preprocessor for C++.

The issue FindClassCandidates starts with a textual `DESCRIPTION` of its purpose, followed by `DECLARATIONS` of local variables. In the example, noun (of type NounArtifact) and NewClass (of type ClassArtifact) are defined along with the string variable ClassName.

The `PROFILE` section contains graphical attributes for the issue, used to control the graphical editor. Only the `DISPLAY` attribute is set, specifying that the graphical object representing the NounArtifact stored in variable noun should be

Figure 12. Architecture of the design assistant

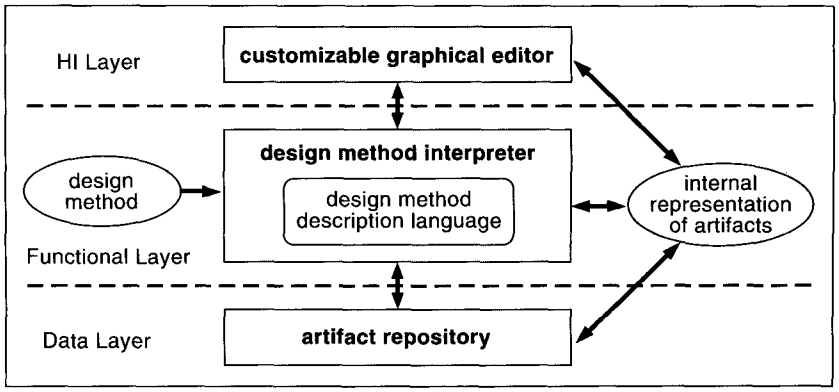


Figure 13. Sample issue from Responsibility Driven Design

```
ISSUE FindClassCandidates IS
DESCRIPTION
        Identify possible candidates for classes from the list of nouns.
        First, select a noun. Then select an argument,
        that supports the fact, that the noun is a class.
END_DESCRIPTION
DECLARATIONS
        NounArtifact*       noun;
        string              ClassName;
        ClassArtifact*      NewClass;
PROFILE
  DISPLAY = noun;

BODY
    FOREACH noun FROM SetOfNouns DO
       SELECT1 ARGUMENT FROM
          Class :      "The noun denominates a physical entity."
          Class :      "The noun denominates a conceptual unit."
          Class :      "The noun denominates a category of terms."
          Value :      "The noun denominates a value of an entity."
          Generic :    "There is an abstract generic term for this noun."
          Unknown :    "None of the above applies."

       DO
         POSITIONS
            Class :    STEP 1; STEP 3;
            Value :    STEP 2; STEP 3; STEP 4;
            Generic :  STEP 2; STEP 3;
            Unknown :  ;

          STEPS
            1.         ClassName = noun->GetName;
            2.         ClassName = ISSUE GetNameOfArtifact;
            3.         NewClass = new ClassArtifact(ClassName);
                       SetOfClasses->Insert(NewClass);
                       SetOfNouns->Remove(noun);
                       delete(noun);
            4.         ISSUE MoreValues(NewClass);
    END
  END
END CLASSCANDIDATES;
```

shown to the user.

The issue BODY contains ARGUMENTS, POSITIONS, and STEPS. In our example, each element of the set SetOfNouns is displayed to the designer, who decides (by selecting one of the arguments) whether and how to create a new class. Arguments may be just textual hints as in this example, or they may be computed in order to check design rules. In our current prototype, the selection of one argument immediately imposes a position (future versions will allow the selection of multiple arguments).

Steps use artifact modification operations. Furthermore, they may call single subissues or collections of issues. In our example, the first three arguments describe the case in which the noun in fact is a ClassArtifact. Therefore, the name of the current artifact noun is used for the ClassArtifact to be created (STEP 1). If the noun is the value of an entity, the subissue GetNameOfArtifact is called in order to get the name of the class (STEP 2). Furthermore, the subissue MoreValues is invoked in order to identify other values of the new class (STEP 4). In all cases a new class is created and inserted into the set of class artifacts (SetOfClasses), and the artifact noun is removed from the set of noun artifacts (STEP 3).

Conclusions

There is much more to distributed object-oriented software development than the provision of distribution-transparent method invocation and object migration support. The DOCASE project tried to develop and support a holistic view of such a development. However, the methodic approach described in this article can only represent one small step toward a more adequate software technology for distributed applications. Many more efforts are needed in this direction in order to avoid a drastic aggravation of the

software crisis as we move from traditional application software to integrated networked solutions in the context of office, manufacturing, and enterprise integration.

The insight into the DOCASE project is of course incomplete. Further activities are the development of a distributed object-oriented runtime system, which is explained by Achauer in this issue [1], configuration support, in particular configuration extensions to DODL [22], an approach to declarative object placement and heuristic object migration support [19], the DODL interaction category, i.e., a typed approach to multiparty communication schedules [20], and the visual programming tool ODE [14]. ◘

References

1. Achauer, B. The DOWL distributed object-oriented language. *Commun. ACM 36,* 9 (Sept. 1993).
2. Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng. 13,* 1 (Jan. 1987), 65–75.
3. Booch, G. *Object-Oriented Design with Applications.* Benjamin Cummings, Reading, Mass, 1990.
4. Cicak, B. and Müller, A. Object-oriented modelling of a production control system. Master's Thesis, Dept. of Computer Science, FH Furtwangen, Germany, 1992. In German.
5. Decouchant, D., Duda, A., Paire, E., Riveill, E., Rouset de Pina, X. and Vandôme, G. Guide: An implementation of the Comandos object-oriented architecture on Unix. In *Proceedings of EUUG Autumn Conference* (Lisbon, Portugal, Oct. 1988).
6. Decouchant, D., Krakowiak, S., Meysembourg, M., Rivelli, M. and Rousset de Pina X. A synchronization mechanism for typed objects in a distributed system. *ACM/SIGPLAN Not. 24,* 4 (Apr. 1989).
7. Gerteis, W. and Wirz, W. Synchronizing objects by conditional path expressions. In *Proceedings of Tools Pacific '91* (Sydney, Australia, Dec. 1991).
8. Gerteis, W., Zeidler, Ch., Heuser, L. and Mühlhäuser, M. DOCASE: A development environment and a design language for distributed object-oriented applications. In *Proceedings of Tools Pacific '90* (Sydney, Australia, Nov. 1990), pp. 298–312.
9. Heuser, L. Processes in distributed object-oriented applications. In *Proceedings of TOOL '90* (Karlsruhe, Germany, Nov. 1990), pp. 281–290.
10. Kaatz, J. Object-oriented modelling of a flexible machine cell. Master's Thesis, Dept. of Electronics, FH Furtwangen, Germany, 1993. In German.
11. Katz, S. A superimposition control construct for distributed systems. Tech. Rep. STP-268-87, MCC, Austin, Tex., 1987.
12. Kramer, J., Magee, J., Sloman, M., Dulay, N., Cheung, S.C., Crane, S. and Twidle, K. An introduction to distributed programming in REX. In *Proceedings of ESPRIT Conference '91* (Brussels, Belgium, Nov. 1991), ESPRIT.
13. Lea, R. and Weightman, J. COOL: An object support environment coexisting with Unix. In *Proceedings of ATUU Convention Unix '91* (Paris, France, Mar. 1991).
14. Leidig, T. and Mühlhäuser, M. Graphic support for the development of distributed applications. In *Proceedings of GI/NTG KIVS '91* (Mannheim, Germany, Feb. 1991).
15. Magee, J., Kramer, J. and Sloman, M. Constructing distributed systems in CONIC. *IEEE Trans. Softw. Eng. 15,* 6 (June 1989), 663–675.
16. Mühlhäuser, M. Computer-based learning with distributed multimedia systems. In *Proceedings of the International Conference on Human Aspects of Computing and Information Management* (Stuttgart, Germany, Sept. 1991).
17. Mühlhäuser, M., Schill, A., Kienhöfer, J., Frank, H. and Heuser, L. A software engineering environment for distributed applications. In *Proceedings of Euromicro '89* Köhn, Germany, Sept. 1989 , pp. 327–332.
18. Potts, C. A generic model for representing design methods. In *Proceedings of the Eleventh International Conference on Software Engineering,* 1989.
19. Schill, A. Mobility control in distributed object-oriented applications. In *Proceedings of the IEEE International Conference on Computers and Communications* (Phoenix, Ariz., Mar.), IEEE, New York, 1990.
20. Schill, A. and Gerteis, W. Communication schedules: An *n*-party communication abstraction mechanism for distributed applications. In *Proceedings of the Tenth ICCC '90* (New Delhi, India, Nov. 1990), pp. 643–651.
21. Wirfs-Brock, R. J., Wiener, L. and Wilkerson. *Designing Object-Oriented Software.* Prentice-Hall, Englewood Cliffs, N.J., 1990.
22. Zeidler, C. and Gerteis W. Distribution: Another milestone of application management issues. In *Proceedings of Tools Europe '92* (Dortmund, Germany, Mar. 1992), pp. 87–99.

CR Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques—*concurrent programming*; D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries*; H.2.4 [**Database Management**]: Systems—*concurrency*

General Terms: Design, Performance

Additional Key Words and Phrases: Concurrency, object-oriented concurrent programming

About the Authors:

MAX MÜHLHÄUSER is professor of distributed systems at the University of Karlsruhe. Current research interests include distributed application engineering, distributed multimedia systems, distributed simulation, and computer-supported instruction and cooperation.

WOLFGANG GERTEIS is a Ph.D. candidate in the Institute of Telematics at the University of Karlsruhe. Current research interests include software engineering, with a special focus on design methods of distributed object-oriented design and programming. **Authors' Present Address:** University of Karlsruhe, Institute for Telematics, D-76128 Karlsruhe, Germany; email: {max, gerteis}@telematik.informatik.uni-karlsruhe.de

LUTZ HEUSER is the director of the applied research and advanced development center at CEC Karlsruhe of Digital Equipment Corporation. Current research interests include distributed programming, workflow-based software engineering, and multimedia collaboration. **Author's Present Address:** Digital Equipment Corporation, CEC Karlsruhe, Vincenz-Priessnitz-Str. 1, D-76131 Karlsruhe, Germany; email: heuser@kampus.enet.dec.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is give that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 0002-0782/93/0900-127 $1.50