Software Engineering in Distributed Systems - Approaches and Issues

Max Mühlhäuser

University of Kaiserslautern, Telematics Group
Erwin-Schroedinger-Str., D-6750 Kaiserslautern, W. Germany

Abstract
Distributed systems come of age, but engineering distributed applications on them remains difficult for the application industry. An adaptation and synthesis of two large areas of computer science is therefore necessary: distributed programming with a focus on object-oriented techniques and software engineering.

The paper tries to give an introduction to, show current trends in, and raise issues of, these areas in the context of distributed application engineering.

The discussion is based on the experience with two large practical software engineering environments developed at the Telematics Groups of the Universities of Karlsruhe and Kaiserslautern. The contributions of the two projects are shown throughout the paper. However, the paper wants to give a classification and state report much more than a report on specific projects, as distributed application engineering is still a rather young field lacking structure and overview.

1 Introduction
This paper reports on issues of development support for distributed applications, i.e. applications intended to run cooperatively on distributed system. An emphasis is put on the issues arising from the distributed nature of such software and on the use of object-oriented techniques, cf. [37,45]. The paper is based on the experiences gained with the development of two experimental software engineering environments (each including a dedicated distributed programming language). The first software engineering environment, called DESIGN, included a distributed extension to C, called DC. DESIGN is operational and is currently used to develop sample distributed applications. A second project, DOCASE, introduces the object-oriented paradigm into our approaches. DOCASE is based on a design-oriented language called DODL and is still under construction.

The problem area is introduced first by describing the typical scenario for a distributed application. This will be done in chapter 2, reflecting the underlying system, distributed (and object-oriented) programming aspects, and software engineering environment aspects. In the following chapter, recent advances in distributed programming, in object-oriented techniques, and in software engineering shall be briefly sketched, along with major research areas of interest as well as our contributions.

2 Principles of distributed applications

2.1 Underlying system scenario
In the nearer future, the development of medium to large computer installations will be driven by the following developments:

- Increasing decentralization through powerful workstations, organizationally integrated in an enterprises flow of work,

- Consolidation of transport subsystem despite the advent of ISDN, HSLANs and multimedia communication, making distributed systems the "standard" computing environment,

- Changing cost-benefit-ratio of remote operations, decreasing the 'penalty' for carrying out operations remotely.

These aspects will lead to a much increased demand for distributed applications. Example application fields comprise, e.g., computer integrated manufacturing, office automation, and globe-spanning management-information systems.

2.2 Distributed application scenario

Transparency considerations: We want to distinguish network transparency, i.e. invisibility of the network's topology and structure and of the physical location of components, from distribution transparency, i.e. transparency of the distributed nature of the underlying system and of the application.

Full distribution transparency is not always the goal of the application programmer. Very often, for example, he may want to make use of his knowledge about the application structure
and behaviour in order to give input to some distribution service (e.g., optimization hints) or for reasoning about the location of users (humans and technical processes) relative to application entities.

Although we said that there will be no overall distribution transparency soon, there will be section wise distribution transparency: the partial use of distributed operating systems, distributed databases, value added networks, and object-oriented distributed languages will introduce partial distribution transparency. As these parts of a distributed system can be regarded, e.g. by distributed applications, like a single huge node, we want to use the term meta-node to comprise distributed operating systems, distributed databases, and value added networks.

Other kinds of transparency in distributed systems are very hard to introduce, such as failure transparency (invisibility of network, node, or software module failures) or performance transparency (lack of visible local / remote performance differences).

**Structural characteristics:** Most problems associated with distributed applications would not vanish given even full distribution transparency. These problems are not exclusively bound to the distributed nature of the underlying system. Rather, they are often present in sequential or concurrent programming already, but become serious along with the use of distributed systems as a basis for application programming:

1. **Structural parallelism** among 'by nature' autonomous and self-contained application elements.
2. **Multiple distributed threads** ('active objects'), obscuring both the difference between heavyweight and lightweight processes and the effects of node boundaries on threads.
3. **Top level complexity** at the level of the active objects with hundreds and thousands of active objects, pertaining to a large number of types.
4. **Irregularity** of the topology, interconnection structure, control flow etc. among active objects.
5. **Dynamics** in the way these irregular structures change at runtime
6. **Asynchronism**, through the extensive exploitation of parallelism.
7. **Multiple conceptual models for communication** (patterns, architectures) within a single large application (message exchange, remote invocation, using point-to-point or multipoint connectivity, etc. [42]).

Figure 1 shows a sample structure small distributed application. The active objects of the distributed application are shaded in black, with the solid lines between them indicating their logical interconnection structure, and the gray-shaded parts indicate 'meta-nodes' (see above).
2.3 Environment / tool scenario

Application distribution: The characteristics of distributed applications as described lead to a number of requirements on the features of software engineering environments and tools:

- Adequate design support: Parallelism, dynamics, and asynchronism support are among the requirements specific for distributed application, graphics / alphanumerics duality, animation support, seamless integration with the implementation language, are some of the general requirements imposed.

- Adequate implementation support: the implementation language, together with tools like source code control and software reuse aids, have to reflect the specifics of distributed applications as well.

- Distribution / installation support: today, standard media copy utilities plus tailored command procedures are mostly used for installation of software products on target systems. For distributed applications, the software engineering environment has to provide tools by which tailored installation support can be prebuilt and centrally controlled via high-level semantics.

- Monitoring / control and data collection / analysis: starting, observing, and managing the execution of an application can largely be performed with operating system utilities in the sequential, single-system case. In the distributed case, corresponding support tools have to be provided by the software engineering / language support. The same applies to collection / analysis utilities for diagnose, optimization, accounting and other purposes.

- Distributed debugging: in the debugging area, the additional problems imposed in the distributed case are conceptually very hard to deal with. The lack of synchronized clocks, the true parallelism leading to (pseudo-) simultaneous events, the deferring influence of debugger software, the information quantity, the additional semantics of distributed applications, all these problems require thorough conceptual consideration [22].

Environment distribution: During the development of distributed applications, the application programmers will most likely work in a distributed system on individual workstations. This means that not only the application is distributed, but also the software engineering environment.

We introduce a distinction of three classes of network nodes (which are not necessarily mutually disjunct):

- Development nodes: these are the nodes on which the development of a distributed application takes place, up to the implementation phase.

- Execution management nodes: the management of the execution of a distributed application is a major task, cf. 2.3. The corresponding management nodes are named here.

- Execution nodes are the ones on which the distributed application actually runs.

Environment distribution, according to our definition, means that the development nodes form a distributed system. If environment distribution takes places, one can further distinguish several levels of sophistication:

- Data distribution: the simplest distributed environments consist of non-distributed environments, where the local database is replaced by a proper database interface on every workstation. In the most primitive case, a single centralized database is offered. More sophisticated of course are truly distributed databases, offering the possibility to store those data local to the software engineer which he accesses relatively often.

- Functional distribution: in a next step, one can 'tear apart' the functional parts of a software engineering environment. In addition to the data layer as above, e.g., the human interaction modules can be placed remote to the proper functions. In addition, different functions can be carried out on different nodes.

- Distributed functions: especially if application distribution takes place, one might think of turning the functions (tools) into distributed programs as well. In compute-intensive areas, most notably in simulation, distributed solutions exists today already. Object-oriented programming will further help to build distributable tools. This way, load sharing can help to make much more efficient use of the potential compute power inherent in a distributed workstation environment.

3 Advances and issues

In this chapter, we want to point to recent advances in the fields discussed, position the specific advances of DESIGN (with its language DC) and DO CASE (with DODL), and to raise open issues and questions.

3.1 Distributed and object-oriented programming

Communication: Concurrent languages (cf. [2,46]) usually assume the existence of shared memory, which leads to inadequate support for remote communication. Truely distributed languages have very much concentrated on the remote communication aspect in the past, providing sophisticated communication architectures and patterns. Some concentrated on synchronous message exchange or more sophisticated extension like rendezvous (DP [12]). Others introduced advanced remote procedure call techniques [7] or atomic, transaction-oriented data exchange (cf. PLITS [24], EDEN [14]). All these patterns tend to restrict parallelism of the communicating active objects by extensive synchronization. The highest degree
of parallelism can be achieved using asynchronous communication (cf. DC [34], Soma [30], DAC [41,19]), or by asynchronous multiple threads of control. However, asynchronism is often strongly discouraged as it hinders 'secure programming'.

As most languages support only one or a few patterns and architectures, they restrict the choice of conceptual models. One solution is to provide a number of predefined packages offering different conceptual models of communication, built on top of asynchronous message exchange (this can be efficiently done, cf. [7] along with [12]). Semantic checks by the language tools and extendibility would have to be provided, too.

Another aspect in this respect is communication abstraction and encapsulation [25,20]. A very notable effort in this direction is the introduction of 'scripts' [26] as a centralized description of the interaction and behaviour of the partners involved in a communication pattern. Communicating active objects can then take the roles offered in the scripts. This approach is taken further in the DODL language of DOCASE [35], where 'communication relations' are offered as part of the top level object hierarchy, introducing top-down structuring and classification capabilities and clear class/instance distinction (Both DODL and [38] suggest to introduce semantics for relations between objects).

Existing distributed object-oriented systems and languages, such as Distributed Smalltalk [6,17,18], EDEN [1], and Emerald [8,9,29] use a single communication model throughout, the method call. Different patterns and architectures, as requested above, are not provided for. The DODL principle of introducing semantic relations - and thereby a hierarchy of predefined communication patterns - represents a step forward in this direction.

Network transparency, even distribution transparency at the level of communication, is offered by virtually all distributed object-oriented languages. Distribution abstraction would however be preferable. DODL therefore offers the choice of explicit distribution information (configured objects) and distribution transparency (dynamic objects).

**Structuring, abstraction, and transparency**: In the remainder, we want to distinguish transparency - the invisibility of an aspect - from abstraction, i.e. the freedom to abstract from an aspect at certain levels of consideration, making it explicit at other levels.

Structuring, abstraction, and transparency are closely related in that structuring is an important base on which abstraction and transparency can be built: hierarchical structuring of entities - one of the most common structuring principles - is a special form of building layers; if the layers within a hierarchy do not exhibit information about deeper layers, then they also form an abstraction principle; and if a user is not allowed to go deeper in a hierarchy than a certain layer, the lower layers are made transparent (of course the difficulty here is that the system has to match higher layers to the hidden ones without user intervention, so that transparency is not just a simple extension to abstraction). Of the various aspects of structuring, the most relevant are code, data, and control flow.

**Code**: Only few approaches for non-object-oriented distributed languages, like DPL-82 [21] and DC, offer hierarchical code structuring at the level of active objects. DPL-82 and DC both follow the black box information hiding principle, allowing processes to create son processes which are invisible to the 'outside world'. Father and son entities can be seen as having a 'subcontracts' relation, i.e. a son entity subcontracts to its father in order to realize some of the functionality exhibited by the father. One of the problems with this type of hierarchical structuring is the question of how to map relations between any 'sibling' entities to their respective 'sons'. DC offers sophisticated ways of mapping communication relations between two active objects to the 'son' active objects hidden by them.

Object-oriented languages offer only three kinds of relationships by default: the 'inherits' relation between classes and subclasses, the 'instanciates' relation between classes and their instances, and the 'calls' relation between a (method-) calling object and a called object. Among these, only 'inherits' is a hierarchical structuring means. Some object-oriented languages offer means for collecting objects in an 'aggregate object', treating them like a single object. 'Aggregates' relations are less powerful than 'subcontracts' relations in that they only offer an 'envelope' around a number of objects, without code or data directly associated to the envelope. DODL is the only object-oriented language we know of which offers a 'subcontracts' relation (via its 'subsystems').

**Data**: With the introduction of object-oriented systems, the differences between data and code begin to vanish, as all data are encapsulated in the code directly manipulating them. The hope is, therefore, that with object-oriented languages, code and data structuring can be realized using the identical principles - maybe the ones just discussed for code structuring. Object-oriented languages and object-oriented databases, however, represent two 'worlds' difficult to merge [43]. This indicates that some severe problems have to be solved if this distinction is to be really removed in the future:

1. Active objects are usually considered as volatile or 'non-persistent', whereas data are often required to be persistent. This distinction between persistent and non-persistent objects has been carried forward to object-oriented technology. A seamless integration, making objects persistent, is currently very difficult for a number of reasons (e.g., operating systems can not be easily built to guarantee persistence of their active objects at any state).
2. If objects are to take the role of data in databases of all kinds, they have to offer the basic access mechanisms applied to data today, namely sequential access (cf. sequential files), and selected access according to index, query, or hypertext search. None of these is usually offered in object-oriented languages, and only either query search or hypertext search are usually offered in object-oriented databases.
Control flow: Major issues of control flow structuring are:

1. In complex applications, especially if they are written using object-oriented techniques, it is hard to trace and understand the control flow. A method call, for example, may represent a 'small sub-function' or the root of a deeply nested, long-term, multithreaded activity. Objects in the narrow sense - encapsulated data structures, together with elementary operations on these data or on other objects - have little in common with a complex control flow superimposed to a set of objects. This means it may be desirable to decouple 'basic code' from control flow. DODL therefore introduces a specific kind of relational object class, the 'cooperation', as the place for complex control flow.

2. Control flow may depend on a number of different aspects, such as the functional aspect of how the application wants to achieve some task, or the operational aspects of how to best carry out an algorithm in a specific installation given the network characteristics. The DODL 'cooperation' relation offers structuring principles for separating these aspects.

3. At execution time, the operating system process used to be the structuring means for control flow, as every control flow was matched one to one to a process, together with the pieces of code associated with the process (linked together in an image). The classical process notion has come to a limit with the need for 'chean' threads in concurrent languages, the need for longliving processes in complex applications (e.g., for modeling a long-duration procedure like that of software production), and the need for node-spanning threads in advanced distributed applications. Once again, the DODL 'cooperation' allows to encapsulate complex control flow and to decouple it from traditional operating system processes.

Finally, as we saw in the introduction to 3.1, structuring principles can serve as one abstraction and transparency means, but they are not the only way to achieve abstraction or transparency of the network structure, distribution and location of entities, performance differences between local and remote operations, and so on. Key open issues for abstraction principles are: what are the more abstract notions for the things one tries to abstract from, and how can they be introduced in a programming or design language / method. Stimuli and responses, pre- and postconditions, resource requirements, and probabilistic descriptions are examples of relatively generic abstractions to consider here.

Administration: The structural complexity described, along with the dynamics inherent in distributed programs, require the application programmer to cope with administration of the (active and passive) objects in a distributed application, both initially and at runtime. Active and passive objects may have to be created, interconnected, reconnected, relocated, terminated etc.

As languages like PRONET [32], CONIC [31], and DC have shown, code structuring helps to cope with the administration problem: e.g., in a hierarchically structured approach, the administration of a set of 'son' entities can be described in a clearly isolated way. Known systems and languages have taken different approaches and have only partly solved the following issues of administration:

1. Administration support at runtime, not just for initial configuration

2. On one hand, clear separation of configuration / administration aspects from functional aspects to reduce complexity and enhance reusability and maintainability,

3. On the other hand, integration of configuration / administration with functional aspects tight enough so that configuration changes can be requested on the basis of functional decisions (e.g., additional active entities in compute bound phases etc.).

Sophisticated support for dynamic administration means that a user can foresee and easily administrate arbitrary numbers of objects at runtime. This means that sophisticated aggregation types (lists, ordered and unordered sets with ordered and contextual access) have to be provided for entities (objects) and their relations.

Integration and extensibility: The complexity of distributed applications makes their development a problem of 'programming in the large'. Along with this, the many development aspects of systematic program development are to be reflected, such as performance, reliability, design, object mobility (cf. [29]), and many others.

The integration of development aspects can never be complete at language definition time. Ideally, one would like to add totally new aspects even to the basic semantics of a language in the course of its use. 'Macro techniques' on one hand, introducing new aspects without semantic support in the language, and recoding of a compiler on the other hand represent the two extremes which both seem to be unacceptable; a compromise in this respect seems to be language extensibility on the basis of highlevel linguistic aids, allowing language extensions to be described to the existing language tools (compiler etc.), and to existing other tools. This way, tools serving new aspects can describe their additions / requirements to the language and language tools. Classical as well as object-oriented languages fall up to now in providing both integration of a large number of development aspects and extension mechanisms, i.e. inclusion of further development aspects after the language is defined: CLU [33] for example integrates the reliability aspect, and DC integrates the performance aspect, but neither is complete or extensible. CENTAUR [11] offers access to a languages semantics at the level of abstract syntax tree, providing complete and runtime extensions in the way tools can interface with the compilation / runtime support.

DODL defines a toplevel object hierarchy which is exported to all tools and which is not modifiable by the application pro-
grammer. Currently, we work on providing integration and extendibility by allowing the toplevel hierarchy to be modified by the system programmer. Ideally, we would like to offer three classes of extensions: 'horizontal extensions' for new semantic relation classes representing a new development aspect; 'lateral extensions' for new toplevel object classes, also representing new development aspects; 'vertical extensions' for new subclasses (using straightforward object-oriented techniques) representing the application domain knowledge, with an ability to be export subclass characteristics to the program development tools.

The most notable aspect to integrate is the design aspect. In addition to the development of more sophisticated object-oriented design methods (cf. [10, 16]), the adoption of some ideas of the widespectrum language approach [5] - offering the same semantical framework from early design to detailed implementation - seems relevant and is pursued in DODL.

Events and Rules: Asynchronism has been recognized as on one hand critical for parallelism and dynamics, on the other hand harmful to secure programming. Longliving activities are driven by asynchronism and by conditions and constraints to a large extend, much more than traditional processes. A formal base has to be established to properly describe all possible actions which may have to be carried out off the track of normal threads. 'Exception handling' facilities as restricted means for describing reactions to a few types of exceptions (offered by DC, *MOD [15], SR [3], and others), seems to be insufficient. A formal base for (asynchronous) events and rules (constraints, conditions), for coupling those descriptions with the normal flow of control - supporting secure programming - has to be developed.

3.2 Environments and tools

Along with this chapter, we want to introduce the basics of the DOCASE coarse architecture as shown in figure 2. The figure is annotated with the issues discussed in the chapter; the terms used will be explained in the remainder.

Integration: As integration of tools is a principle goal of any software engineering environment, one could expect to find sophisticated integration approaches to be used in state of the art environments.

As large software engineering environments will always have to be extended in their lifetime, extendibility could be expected to be focused on. Looking at the three almost 'classical' layers of software engineering environments, the human interaction layer, functional, and data layer, one finds that the integration, as well as extendibility, are still problems which are largely unresolved (compare, e.g., [40] to [44]). A principle problem lies in the fact that environments try to integrate existing tools of different age produced by different groups or vendors; the tools are usually not extendible or flexible enough to be subordinated to a new overall method or philosophy.

1. The human interaction layer has in the past not been

![Diagram of DOCASE architecture and issues](image-url)
clearly separated from the functional layer. 'Common look' appearance of different tools of different sources has therefore hardly been achieved. The decoupling of human interaction and functional layer has recently been pushed with the upcoming workstation windowing system standards, most notably X-Windows [40]. These standards are however on the level of graphics primitives and not on the level of a standardized software engineering layout philosophy and related screen artifacts (such as screen representations of active entities, class/instances, messages, calls, modules, statements etc.). Such a 'software screen artifact standard' approach, together with a standardized protocol for the communication between tools and the human interaction layer, could insure a high level of common look appearance for tools of different sources, but for the moment is not in sight. Another, mostly complementary approach is that of mapping generic 'common look' screen artifacts to the input of different tools and mapping their output back again. This way, existing tools could remain largely unchanged. Recent efforts in the direction of such a 'mapping' approach are made in the INCAS project. The mapping process is however difficult to achieve, as not only artifacts but philosophies and methods have to be matched, a problem which has to be dealt with in the data layer (see below). DOCASE tries to mix the approaches partly, defining its 'private' software screen artifact standard and adding a limited mapping functionality (the mapping can remain limited due to the fact that DOCASE uses a self defined language, DODL).

2. As most environments were built around existing, mostly inflexible tools, integration at the functional layer has always been a difficult task. The recent advances of CASE tools [28] brought up approaches in which a suite of tools, a toolset, was designed in a single project, around a harmonized set of methods, i.e. with a common feel appearance in the functional layer. Several toolsets focused on the early program development phases; the last output of the toolset was skeletal code for standard programming languages (in this context, we want to disregard 4GL languages because of their still very restricted application domains). The idea was to have the programmer fill out the skeletons, and the idea was also that using this technique one could use the - typically very elaborate - compilers, runtime systems, and debuggers of different target machines. However, in every environment in which the programming language is still visible to the programmer (i.e., where he cannot stay within the semantics of a higher notation throughout every phase of the lifecycle), this programming language remains the center in the lifecycle around which upstream activities (analysis, design etc.) and downstream activities (testing, debugging, measurement etc.) are centered. The existing language tools are usually inflexible and unextendible, and therefore a 'common feel' integration of upstream and downstream activities around their center, the programming language with its tools, is unfeasible.

As discussed in 3.1, DODL tries to serve as both a design and an implementation language, making a predefined object hierarchy both visible to the tools and extendible.

3. In the data layer, two basic approaches to integration are widely discussed [23]: the 'common object' approach forces tools to use a common object repository. This approach has found much interest recently and has entered the standardization process with a proposal called ATIS, it is however not properly feasible if older tools - not following the standard - have to be included. A 'mapping' approach, analogous to that described for the human interaction layer, allows the integration of existing tools, but requires numerous data translation functions for the mapping and lets the data repository centrally define only the structure of the data, but not their semantics; if one tries to avoid a central data repository for the environment, data translation has to be inserted between any possibly connected tools.

Central repositories surely have a large number of advances, like databases in general have for large applications. Standard databases, however, have long been recognized as insufficient for software engineering environments, as mentioned in [43,36]. The database model has therefore evolved in the past from the classical relational model via entity-relationship and entity-relationship-attribute models to the object model. The latter will surely stay the preferred model in software engineering environments for the near term future. However, 'vertical' integration with the functional layer into 'persistent active objects' (see above), and 'horizontal' integration in the sense of hypertext search (in addition to query search) will take some time to become mature. The different characteristics of software bases as opposed to classical data bases (e.g., lower access rates to a higher number of objects) have led to special considerations of 'object software bases' [4].

Aspect separation: Distributed applications introduce a number of development aspects in the software engineering process which are unknown in sequential, single node applications, such as object placement, parallelism enhancement, and fault tolerance. Other aspects become much more relevant and/or more difficult to deal with, such as administration support and performance optimization. The large number of aspects, and the thread of an even aggravating software crisis, makes it extremely important to be able to deal with a single aspect at one time. This way, experts for different aspects can work on a project, and complexity is reduced. At the same time, one would like to deal with an aspect in different phases of the lifecycle, and do this in a 'common look and feel' sense. This leads to the requirement of building a comprehensive workbench for every aspect, which each workbench spanning the lifecycle (to the extend the aspect spans it), and offering common (user-level) conceptual models, vocabulary and interaction concept (screen design). (cf. 2.2, 3.1, workbenches can help introduce transparency / abstraction for the development aspects). The notion of workbenches as defined here is used in DOCASE.

Environment distribution: Current standardization efforts in Europe (PCTE [27]) and the US (CAIS [39]) already consider environment distribution. These efforts tend to standardize a
'shell' for environments, consisting of basic elements of the human interface layer (e.g., X-Windows), the functional layer (tool building tools), and the data layer (offering some framework according to the 'common object' approach). We prefer the term 'shell' to the commonly used term 'kernel' as we have a different view of an environment architecture (cf. fig. 2). The environment distribution in the standardized shells is restricted to data distribution as described in 2.3. In addition, they do not consider the clear distinction of node types as proposed in 2.3 (to our knowledge, only DESIGN and DOCASE reflect such a distinction in the conceptual framework of the environment). A number of environments are built on top of standardized shells (e.g., Eclipse [13]), but those are related closely to the specific standard they choose.

Tool adaptation: A large number of individual problems have to be solved in order to find new and enhanced concepts for tools suitable for distributed application development. The major areas to consider have already been described in 2.3. We cannot go into detail here for the specific areas.

This seemingly unstructured field of 'tools suited for distributed application developments' might look different if one came to a common understanding of the distributed application development process, and from there to a distributed application engineering methodology. Tools adhering to this process might then be more easily integrated and their problems might be understood and solved in a broader context. The lack of a common methodology - even of a common vocabulary and paradigms - for many fields of software engineering has made this field of computer science too much an 'art' instead of an 'engineering process' or even a science.

4 Summary

We have looked at the mutual influences and alternating effects of distributed application engineering on one hand and distributed programming languages, object-oriented techniques, and software engineering on the other hand. The major relevant issues in this respect have been raised, many of which are addressed in the DESIGN and DOCASE projects. The paper is meant as a first step towards a common understanding about those major issues, leading to new approaches and, in the long run, to a common distributed application engineering methodology.

5 References


[23] Falk, H.:
Software vendors serve up varied palette for CASE users
Computer Design, 111 1989, pp. 70 - 80

High Level Programming for Distributed Computing,
CACM Vol. 22 No. 6, 1979, pp. 353-368

[25] Forman, I.R.:

[26] Francez, N., Halpern, B.:
Script: A Communication Abstraction Mechanism,
Operating Systems Review, No. 2., April 1985, pp. 53-67

[27] Gallo, F.:


[29] Jul, E., Levy, H., Hutchinson, N., Black, A.:
Fine-Grained Mobility in the Emerald System

[30] Kessels, J.L.W.:
Soma: A Programming Construct for Distributed Processing
IEEE Trans. SW Engineering, 7/1, 1981, pp. 502 - 509

CONIC: an integrated approach to distributed computer control systems. IEEE Proc., 130/1, January 1983

[32] LeBlanc, R.J., Maccabe, A.B.:

[33] Liskov, B.:
On Linguistic Support for Distributed Programs

[34] Mühlhäuser, M.:

[35] Mühlhäuser, M., Schill, A., Heuser, L.:

[36] Penedo, M.:

[37] Rentzsch, T.:
Object-Oriented Programming.
ACM SIGPLAN Notices, vol. 17, no. 9, Sept. 1982

[38] Rumbaugh, J.:
Relations as Semantic Constructs in an Object-Oriented Language. Proc. OOPSLA 87 12/87, pp. 466-481

[39] Scheffer, P.:

[40] Scheffer, R., Gettys, J.:

[41] Staroste, R., et. al.:
A Portability Environment for Communication Software

[42] Tanenbaum, A.S., van Renesse, R.:

[43] Tsichritzis, D., Nierstrasz, O.:
Fitting Round Objects Into Square Databases

[44] Wasserman, A., Fincher, P.:
A Graphical, Extensible Integrated Environment for Software Development

[45] Wegner, P.:
Dimensions of Object-Based Language Design
OOPSLA '87 Proceedings, ACM 1987

[46] Yokote, Y., Tokoro, M.:
Experience and Evolution of Concurrent Smalltalk,
OOPSLA '87 Proceedings, ACM 1987