Current Trends: Distributed Programming - Software Engineering - Object-Oriented Techniques

Dr. Max Mühlhäuser

University of Karlsruhe, Institute for Telematics
Zirkel 2, D-7500 Karlsruhe, W. Germany
Tel. (+49) 721-608-3391

It is a common complaint in the computing departments of medium to large companies these days: they cannot sufficiently meet the demand for ever higher integration of their application software into computer integrated manufacturing systems, distributed office automation systems, enterprise-wide information management systems etc. While the hardware requirements can be largely met with state-of-the-art distributed system technology, no sufficient concepts for development and integration of distributed application software are available.
We believe that a synthesis of three large areas of computer science is necessary in order to counterface this problem: distributed programming, software engineering, and object-oriented techniques.
The contribution tries to give an introduction to and show current trends in the three areas, as an introduction to the fields covered by the workshop sections. Open issues and approaches to a synthesis of the three areas are sketched in order to demonstrate the purpose and leading topic of the workshop.

1 Introduction

This paper is intended as an introductory overview of the workshop topics, leading to the contributions in the sessions which follow. More precisely than by 'distribution and objects', the central theme for the workshop can be called 'support for the construction of complex software, with a focus on both the distributed nature of such software and on the use of object-oriented techniques (cf. [REN82, WEG87]) for supporting their construction'.

In this context, we would first like to introduce the problem area by describing the typical scenario for a distributed application we have in mind. 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 open questions to be possibly elaborated in the workshop.

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:

- o **Increasing decentralization:** the workstation generation which currently replaces the PC generation of systems will represent a major push towards distributed systems: their increased power and easy network integration makes workstations feasable for performing integral parts of an enterprises data processing on them; while PCs were largely used for isolated tasks such as desk top publishing or for sophisticated terminal emulation, workstations are about to become an integral part of an enterprises computing system and of the enterprise-wide data processing procedures.

- o **Consolidation of transport subsystem:** The establishment of the OSI protocols has laid a common ground for the construction of large heterogeneous networks. This robust, well-established, and well-understood transport-oriented protocol stack represents another major step towards making distributed systems widely usable for application programmers.

  It has to be mentioned, however, that the further introduction of the OSI protocols will be restrained by a number open issues to be resolved:

  - Public networks evolve towards ISDN and the issue of integrating the OSI and ISDN protocols and networks is still not totally resolved.

  - With the introduction of fast communication media, the effort necessary for message processing in the 'protocol stack' - formerly an overhead of a small percentage - has become the gating factor and bottleneck.

- For the forseeable future, neither communication capacities nor communication patterns will consolidate: capacities beyond those of FDDI and Broadband-ISDN are investigated, and communication patterns like those of individualized interactive multimedia applications continue to influence the view of a 'typical' network load.

o **Changing cost-benefit-ratio of remote operations:** while the power of individual network nodes - especially workstations - is often said to be exponentially growing, the communication capacity of typical networks is growing even faster: the introduction of dedicated cabling (mainly coax) with the first generation of LANs and the recent - and ongoing - introduction of fibre optics cabling in both LAN and WAN installations is about to drastically decrease the 'penalty' for carrying out operations remotely. This, again, makes distributed programming much more attractive.

These aspects will lead to a much increased demand for distributed applications. The application domains for complex, highly distributed applications can hardly be forecasted because today's imagination is too much bound by the application structures and domains we know today. Some example fields can however be cited today already, and in special areas, e.g., computer integrated manufacuring, office automation, or globe-spanning management-information systems, many of the problems discussed in this article have become the daily bread of software engineers already.

2.2 Distributed application scenario

2.2.1 Network transparency and partial distribution transparency

In this context, 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. One might forsee for the future that after a short transition period, distributed programming would lose its mystery due to the introduction of distributed operating systems, sophisticated object oriented support systems, and/or powerful name services; these would offer both network and distribution transparency; and the user could - at the sematic level - continue to use traditional programming languages, at most concurrent programming languages.

Distribution transparency is however in many cases not 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., the runtime system of the programming language) about how to optimize application performance by distributing application components. Or it may be that users (humans and technical processes) located at different sites have to be served and the fact that they are not all on one system has to be reflected in the application structure. Overall distribution transparency is therefore not likely to be *the* common ground for the programmer to build on in the mid term future, while network transparency is likely to become mature very soon.

Although we said that there will be no overall distribution transparency soon, there will well be sectionwise distribution transparency: first, some local clusters of nodes will be part of distributed operating systems - offering distribution transparency among the active objects within that cluster; second, some nodes will also offer distributed databases, offering distribution transparency among the related data; and third, value added networks will evolve to powerful 'network machines', offering distributed services to the user with distribution transparency as well. These three developements will however, as stated, only lead to large and powerful 'meta-nodes' in a larger distributed system, with distribution transparency offered *within* every meto-node.

2.2.2 Structural characteristics

Most problems associated with distributed applications would not vanish given even full distribution transparency. This is due to the fact that many of these problems are not exclusively bound to the distributed nature of the underlying system. Rather, they are often present in sequential or concurrent programing already, but become visible and serious along with the use of distributed systems as a basis for application programming. The main characteristics and problems which deteriorate with distributed application programming are listed below:

a. *Structural parallelism:* we consider most of the parallelism exploitable in a distributed application to be structural ('coarse grained'), i.e. introduced in the process of mapping different, largely independent objects of the real world to - again largely independent and by nature parallel - program entities. Communication and synchronization of structurally parallel objects are introduced to have these objects contribute in synergy to an overall goal. We distinguish structural parallelism from rather functional ('fine grained') parallelism; the latter term means that a selfcontained algorithm or operation - which in most cases could even be described much more easily in a sequential way - is 'parallelized' for computation speedup. The distinction between structural and functional parallelism may seem to be philosophical, but it is in fact largely the reason for the further characteristics listed here and in this role presents the major reason why distributed applications are and will be much different from typical concurrent programs as known today; a reason, once again, which is not bound to distributed applications, but characteristic for them.

b. *Multiple distributed threads:* parallelism is typically achieved via multiple threads. Many concurrent implementations offer multiple threads as 'lightweight processes' within a single operating system process. Many distributed applications were realized in the past via multiple heavyweight processes, each containing a single thread, and with every process bound to a computer. More recent developments, especially along with object-oriented techniques, tend to hide implementational considerations such as whether to use a lightweight or a heavyweight concept in one or the other case. An 'active object' is allowed to contain several ('fine grained') threads, and to span multiple nodes.

c. *Toplevel complexity:* It is at the level of these 'active objects' that distributed applications show a high degree of complexity. Hundreds of active objects, pertaining to tens of types, are not unusual already in present advanced distributed applications. Information hiding and top-down structuring principles are, however, often introduced within single active objects only: in 'classical' languages, these are the modules, procedures, functions etc. which a single process may consist of.

d. *Irregularity:* while functional parallelism is very often regular in nature (cf. recursiveness of divide-and-conquer algorithms or multiple parallel data streams as used in many image manipulation algorithms), structural parallelism is typically much less regular. This increases complexity and decreases visualizability and comprehensiveness inherent in distributed applications.

e. *Dynamics:* characteristics like the workload input to a distributed application, the network nodes involved, and the number of users logged in to the application, typically vary substantially not only from one execution to the other, but also within runtime of a single execution. In a distributed application, one wants to or has to react to these dynamic changes by dynamic changes of the application topology (number and interconnections of active objects)

f. *Asynchrony:* if parallelism is to be exploited extensively, this increases the number of potential asynchronous events to consider. In sequential applications, runtime errors are often the only asynchronous events possible, and these are often disregarded by the programmer. A distributed application programmer has to face a higher probability of runtime errors due to the reduced reliability of the overall distributed system (total failure transparency not being expected to be commonly available soon), and he has to reflect asynchronous events generated by the different active objects running in parallel.

g. *Multiple conceptual models for communication:* in large applications, the active entities often follow quite a number of different communcation *patterns* and *architectures* [FOR86]: *patterns* like message exchange, remote invokation, or atomic transaction-like data exchange using point-to-point or multipoint connectivity, in synchronous and asynchronous variants, within client/server, actor, or balanced *architectures*, and many many more. A single architecture and pattern tends to bind the user to a single model in an untolerable way [TAN88].

Figure 1 shows a sample structure of a comparatively small distributed application in an environment with partial distribution transparency, trying to demonstrate the characteristics described in 2.2.1 and 2.2.2. 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'.

2.3 Environment / tool scenario

2.3.1 Application distribution

The characteristics of distributed applications as described in the above section lead to a number of requirements on the features of adequate software engineering environments and tools. Some major requirements are listed below:

o *Adequate design support:* the design methods common for sequential, even for concurrent programming are inadequate for distributed applications. Most of the characteristics and

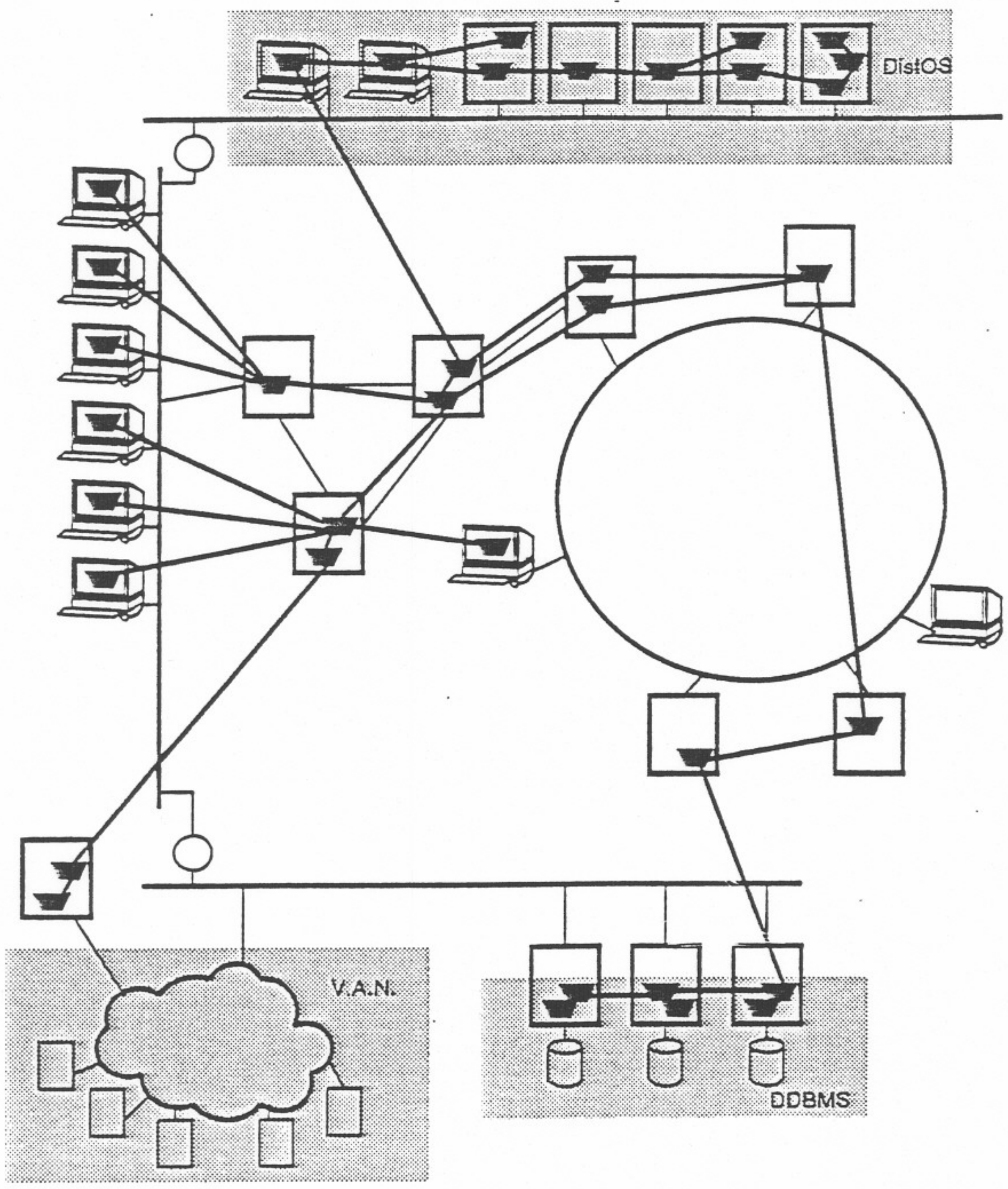

Figure 1: Example distributed system scenario with example distributed application

problems mentioned in 2.2.2 and 2.2.3 are not sufficiently reflected there. New design methods and adequate tools have to be introduced. Simultaneous support for graphical and alphanumerical design, animation support, seamless integration with the implementation language, are some of the requirements imposed on *any* design method today, which have to be met by design methods for distributed applications as well.

- o *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. As we will see in chapter 3, existing languages provide only point solutions to the proble~~ms~~ mentioned in 2.2.

- o *Distribution / installation support:* today, standard media copy utilities plus tailored command procedures are mostly used for installation of software products on target systems. The installation of distributed applications on a whole network cannot be carried out efficiently using such techniques. Rather, the software engineering environment has to provide tools by which tailored installation support can be prebuilt. Such installation support is to be executed on the target system, providing central control and high-level semantics for the installation of the application on the target network, including, e.g., parameter file distribution, network-wide logical name definition, etc.

- o *Monitoring / control and data collection / analysis:* starting, observing, and manageing the execution of an application can largely be performed with operating system utilities in the sequential, single-system case. In the distributed case, corresponding and suitable support tools are normally not offered by the network operation / management software. They have to be provided in the software engineering environment. Collection and analysis of data about the application, for diagnose, optimization, accounting and other purposes, is another field which is (although to a lesser extent) supported by operating system utilities in the sequential case; once again, the environment must provide appropriate support in the distributed case.

- o *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 deffering influence of debugger software, the information quantity, the additional semantics of distributed applications, all these problems require thorough conceptual consideration. In fact, we regard distributed debugging as one of the most difficult problems in distributed application development [FAG88].

In addition to the problems and requirements imposed by the distributed nature of the applications considered, there are requirements which again are present already in the sequential case, which just become more important in the distributed case. Among them are 'programming in the large' requirements like sophisticated project management, seamless tool integration, or 'environment distribution' as described below.

2.3.2 Environment distribution

Developing (and, e.g., testing) distributed applications, the application programmers will most likely work in a distributed system themselves. Being highly qualified engineers working in a large software project, they will most likely dispose of individual workstations.. But this means that not only the application is distributed, but also the software engineering environment.

Along with environment distribution, we introduce a distinction into three classes of network nodes (which are not necessarily mutually disjunct):

- o *Development nodes:* these are the nodes on which the development of a distributed application takes place, up to the point when the code is ready to be transferred to the target execution network(s) (note that this network is not necessarily different).

- o *Execution management nodes:* as stated in 2.3.1, the management of an execution of a distributed application (distribution / installation, monitoring / control, data collection / analysis, debugging) is a major task. The nodes from which such execution management is steered are called execution management nodes.

- o *Execution nodes:* obviously, the application will be executed on a distributed system as well. The respective nodes are called execution nodes. They have to be online connected to the execution management node(s), but not necessarily to the development nodes.

Environment distribution, according to our definition, means that the development nodes form a distributed system. Note that environment distribution makes sense even if application distribution is not intended.

If environment distribution takes places, one can further distinguish several major categories:

- o *Data distribution:* with the above metioned 'typical' environment in mind - groups of software engineers with individual workstations, working on one project - the relevant objects shared within the distributed environment are the development artifacts, i.e., data. Therefore, a first approach to environment distribution is to install non-distributed environments with a proper database interface on every workstation, and to exchange the local database by a distributed one. In the most primitive case, only remote access to centralized data is offered. More sophisticated of course are truely distributed databases, offering the possibility to store those data local to the software engineer which he accesses relatively often.

- o *Functional distribution:* in a next step, one can look at the coarse functional structure of a software engineering environment, distinguishing the human interaction layer (HIL), the functional layer (the tools, in essence), and the data layer. Along with the workstation era, HILs tended to be clearly separated from the functional layer; it was therefore an obvious step to distribute the HIL and the functional layer among different nodes. This way, both compute-intensive functions and graphics-intensive HIL features can be put on the appropriate network nodes. In addition, different functions can be carried out on different nodes. To summarize, with functional distribution, the HIL layer resides on the users workstation, and functions are carried out potentially remotely.

- o *Distributed functions:* especially if application distribution takes place, one would think of turning the functions (tools) in an environment into distributed programs as well. In compute-intensive areas, most notably in simulation (used in software engineering environments, e.g., for rapid prototyping and performance evaluation), 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, and, more important, we want to raise open issues and questions which we hope to find a common understanding about in the workshop. As a basis for discussions in the workshop, some paragraphs of the remainder will be shortly rephrased as 'issues'.

3.1 Distributed and object-oriented programming

3.1.1 Communication

Concurrent languages, like Concurrent Pascal or ADA ([AND83]), have been designed under the assumption that concurrent active objects can share common memory. This has lead to a focus on communication and synchronization models like monitors, which can hardly be efficiently implemented in a distributed system. (Moreover, those languages did hardly address any of the characteristics and problems addressed in 2.2.2 and 2.2.3.). Concurrent object-oriented languages (cf. Concurrent Smalltalk [YOK87]) have put less emphasis on structural parallelism than required for efficient distributed programming.

Truely distributed languages have very much concentrated on the communication aspect in the past, i.e., on the provision of sophisticated communication archticetures and patterns. Some concentrated on synchronized message exchange or its more sophisticated extension, the rendezvous (DP [BRI78]) . Others introduced advanced remote procedure call techniques ([BIR84]) or atomic, transaction-oriented data exchange (cf. PLITS [FEL79], EDEN [CAL87]). One problem with all the patterns mentioned here is that they 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 [MÜH88], Soma [KES81], DAC [EBE88, STA88]), or by multithreaded active objects where only the communicating threads have to be blocked for synchronization (this second approach is sometimes controverse to the programmers conceptual model and therefore is not very common). On the other hand, asynchrony is often strongly discouraged as it hinders 'secure programming'. Another problem lies in the fact that most languages support only one or a few patterns and architectures, restricting the choice of conceptual models. It has been shown that asynchronous message exchange, again, might be of great help because any other model can be efficiently built on top of it (cf. [KES81] along with [BRI78]). Both point-to-point and multipoint connectivity have to be supported in this case. The ideal solution to imagine would be a number of predefined packages offering different conceptual models of communication, built on top of asynchronous message exchange. Semantic checks would have to be offered by the language tools, and the addition of new models would have to be possible.

Another aspect in this respect is communication abstraction and encapsulation [FOR86, ELR82]. A very notable effort in this direction is the introduction of 'scripts' [FRA85] 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 [MSH88], where 'communication relations' are offered as part of the toplevel object hierarchy, introducing top-down structuring and classification capabilities and clear class / instance distinction (Both DODL and [RUM87] suggest to introduce semantics for relations between objects).

Distributed object-oriented systems and languages, such as Distributed Smalltalk [BEN87, CUL87, DEC86], EDEN [ALM85], and Emerald [BLA86, BLA87, JUL88], use a single communication model throughout, the method call. As the conceptual model of the user may range from the data exchange paradigm to the remote invokation paradigm (which method call is closest to), and as multipont communication, different levels of synchronization, atomicity and other aspects cannot be sufficiently reflected, plain method call is obviously not sufficient. Once again, 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* - offering access to distribution information at the level desired, but also abstracting from it at higher levels - is however preferrable if a communication mechanism wants to make use of the distribution information.

Issue: how to construct a flexible communication mechanism, supporting a large number of conceptual models with semantic checking, providing extendability for new models, offering security and asynchrony at the same time, offering abstraction and encapsulation, providing network transparency and distribution abstraction.

3.1.2 Structuring, abstraction, and transparency

Manageing the complexity of distributed applications is largely related to the availability of powerful structuring, transparency, and abstraction mechanisms. The toplevel complexity of distributed applications, as described above, can be partly faced with sophisticated *structuring* principles, such as information hiding, top-down and bottom-up design concepts. Examples for *transparency* (network transparency, distribution transparency, and failure transparency) have been shortly introduced already. In distributed applications, more areas exist in which one would like to make characteristics or structures invisible to the programmer (*transparency*) or to allow him to abstract from such at certain levels of consideration (*abstraction*): performance transparency (local and remote calls performing nearly identically in object-oriented systems) and communication abstraction (the ability to abstract from details of communication patterns while designing an active object) may serve as two further examples.

Structuring, abstraction, and transparency are closely related: hierarchical structuring of entities, being one of the most common structuring principles, is a special form of building layers; if the layers within a hierarchy do not exhibit any 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 an easy add-on to abstraction). Of the various targets of structuring, the most relevant are *code*, *data*, and *control flow*.

Code: Of the numerous approaches for non object-oriented distributed languages, only few like DPL-82 [ERI82] 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 in this type of hierarchical structuring is the question of how to map relations between any 'sibling' entities to their respective 'sons'. PLANET and DC offer 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 'instantiates' 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 aggregating objects within an 'aggregate object', treated like a single object by other objects. '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).

Issue: Which code structuring principles are to be offered.in order to help coping with the complexity inherent in distributed applications; is there a formal foundation which would allow to commonly express different structuring principles and which might show to what extend a set of principles is 'complete' or 'overlapping'.

Data: With the introduction of objects-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 [TSI88]. This indicates that some severe problems have to be solved if this distinction is to be really removed in the future:

a. Active objects are usually considered as volatile or 'non-persistent', whereas data are often required to be persistent. This distinction of persistent and non-persistent objects has been carried forward to object-oriented technology, and a seamless integration, e.g., by making every object persistent, is currently very expensive since, e.g., operating systems can not be easily built to guarantee persistence of their active objects at any state.

b. 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 acceess 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.

c. 'Classical' databases on one hand have proven to be badly applicable to many present problem domains (e.g., software engineering environments); on the other hand, in their strong domains they offer a functionality and efficiency not yet accomplished by any other means. So, in order for 'classical' databases to be replaced by persisten object systems, the latter have to perform at least as good as the former in all domains.

Issue: is a 'persistent object system', integrating object-oriented programming language and database functionality, economically feasable in the near term, and if so, in which domains is it really advantageous over programming languages and databases, respectively.

Regarding the code as the formal description of contol flow, one could argue that code structuring and control flow structuring are in principle the same. Such a statement, however, neglects several major facts:

a. 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, longterm, 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.

b. 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. One may want to apply structuring principles for separating different such aspects.

c. 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 'cheap' 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, a new specific structuring means for encapsulating control flow, both at implementation time and at execution time, seems to be necessary.

Issue: Should a specific means for expressing and structuring control flow be introduced, and how should it look like?

Finally, as we saw in the introduction to 3.1.2, 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. A key question for abstraction principles is: what *are* the more abstract notions for the things one tries to abtract from? Stimuli and responses, pre-and postconditions, resource requirements, and probabilistic descriptions are examples of relatively generic abstractions. An issue which we want to totally defer to the workshop discussions is therefore

Issue: Which kinds of abstraction, which kinds of transparency are most desirable, which abstract terms exist to describe them?

3.1.3 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 passice) 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 [LEB82], CONIC [KRA83] 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 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:

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

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

c. 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 forsee - 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.

Issue: how can encapsulated configuration support *and* close interaction with functional code be achieved; how to associate the active object paradigm and the persistence paradigm to configuration support; are aggregation types the only prerequisite for sophisticated support of arbitrary dynamic changes?

3.1.4 Integration and extendability

The complexity of distributed applications makes their development a problem of 'programming in the large'. Distributed application programming carried out by only one or two programmers can hardly be imagined. Along with this, the many *development aspects* of systematic program development are to be reflected, such as the performance aspect (optimizing the application performance), the reliability aspect (building in fault tolerance), the design aspect (e.g., coupling design artifacts - formal or graphical design, documentation etc. - with the corresponding implementation artifacts - code -), the object mobility aspect (cf. [JUL88]) 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; the 'ideal' language/tool feature in this respect seems to be language extendability on the basis of highlevel linguistic aids. This way, changes could be described to the existing language tools (compiler etc.), and to existing other tools. New tools could easily describe their additions / requirements to the language and language tools. Both 'classical' and object-oriented languages fail up to now in providing *integration* of a large number of development aspects and allowing *extension*, i.e. inclusion of further development aspects after the language is defined. CLU [LIS82] for example integrates the reliability aspect, and DC integrates the performance aspect, but neither is complete or extensible. CENTAUR [BOR88] offers access to a languages semantics at the level of the abstract syntax tree, providing compiletime 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 programmer. Currently, we work on providing integration and extendability by allowing the toplevel hierarchy to be modified by the system programmer. Ideally, we would like to offer three classes of extensions: 'horizontal extensions' would denote adding new semanitc relation classes when adding a new development aspect; 'vertical extensions' would allow to make subclasses of the toplevel hierarchy (defined by straighforward object-oriented techniques) part of the toplevel hierarchy, thereby allowing generic domain knowledge to be exported to the program development tools; 'lateral extensions' would modify existing and add new methods to existing toplevel object classes.

Issue: which are the pathways towards integration and extendability; are the research fields 'language extendability' and 'formal semantics' in a stage where integration and extendability in our sense can be reached; are our three dimensions of extendability 'spanning the space' for extendable object-oriented systems?

The most notable aspect to integrate is the design aspect. In addition to the integration as described above, the adaption of a widespectrum language approach [BAU82] - offering the same semantical framework from early design to detailed implementation seems relevant. Although object-oriented programming has proven to be of great benefit to the design aspect (cf. [BOO86], [COX86]), the lack of explicit semantic differences between objects and the lack of integration of design and implementation languages have not been resolved very well in the past.

3.1.5 Events and Rules

Asynchrony 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 asynchony 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[COO80], SR[AND82], 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.

Issue: how to get to a formal base for a flexible and secure handling of events and rules?

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

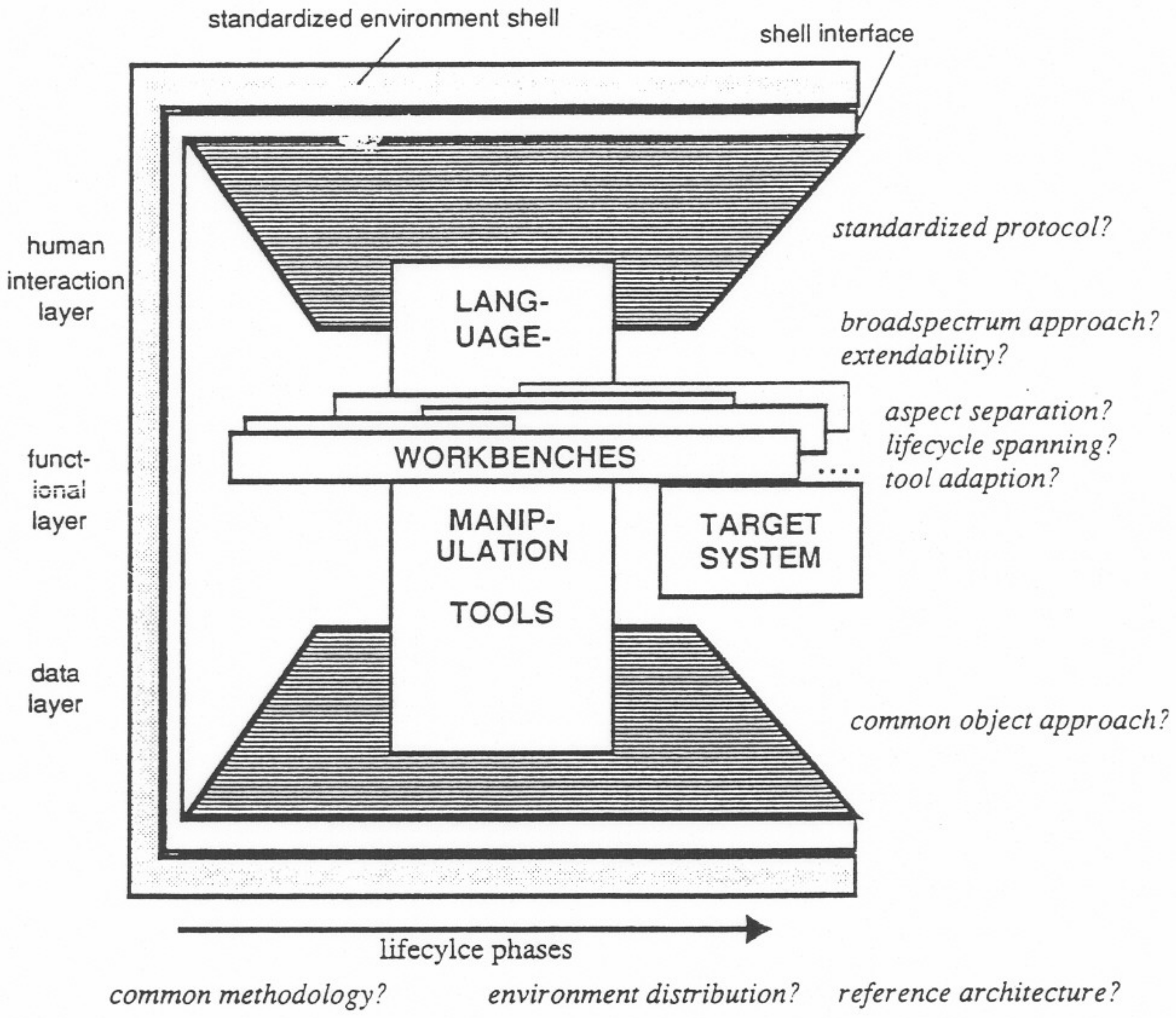

Figure 2: Coarse DOCASE architecture and issues

will be explained in the remainder.

3.2.1 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, extendability could be expected to be focussed on. Looking at the three 'classical' layers of software engineering environments, one finds that the integration, as well as extendability, are still problems which are largely unresolved (compare, e.g., [SCH84] to [WAS86]) . 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 extendable or flexible enough to be subordinated to a new overall method or philosophy.

a. The *human interaction layer* has in the past not been 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[SCH87]. 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, classes/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 selfdefined language, DODL).

   Issue: to what extend is the 'softwae screen artifact standard' and the 'mapping' approach feasable; how else can 'common look' be achieved?

b. 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 [IEE88] 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 (of course this was a major step towards integration in the HI and data layers as well). Several toolsets focussed on the early program development phases; the last output of the toolset was sceletal 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 lifecylcle around which *upstream* activities (analysis, design etc.) and *downstream* activities (testing, debugging, measurement etc.) are centered. The existing language tools are usually inflexible and unextendable, and therefore a 'common feel' integration of upstream and downstream activities around their center, the programming language with its tools, is unfeasable.
   As discussed in 3.1.4, DODL tries to combine recent advances in widespectrum languages with openness and extendability. As the language tools are accessable and extendable, tight integration with both upstream and downstream activities seem feasable.

   Issue: have broadspectrum language approaches and extendable language approaches reached enough maturity to be used in a software engineering framework? Can they be applied to distributed languages?

c. In the *data layer*, two basic approaches to integration are widely discussed [FAL89]: 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 feasable 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 [TSI88, PEN86]. 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' [ARA88].

   Issue: which are the data base approaches and models mature enough to build environments on, which are the forthcoming ones, and how can we protect the investment in building functional and human interface layers on top of changing database technology

3.2.2 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). The notion of workbenches as defined here is used in DOCASE.

Issue: how to find common principles, architectures, or even a formal ground for the process of developing and integrating workbenches for software engineering environments.

3.2.3 Environment distribution

Current standardization efforts in Europe (PCTE [GAL87]) and the US (CAIS [CAS87]) 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 2.3.2. In addition, they do not consider the clear distinction of *node types* as proposed in 2.3.2. (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 [CAA87]), but those are related closely to the specific standard they choose.

Issue: can the different types of environment distribution and the distinction of node types be elaborated into a reference architecture for distributed software engineering environments, and can standardized environment shells be commonly interfaced to?

3.2.4 Tool adaption

It seems as if 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 2.3.1. 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 current 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.

Issue: How big and of what kind is the influence of distributed application development on the major tool areas (development aspects); is there a path towards a commonly agreed upon methodology for distributed application development; can we help to enforce its development?

4 Summary

We have looked at the mutual influences and alternating effects of distributed application programming on one hand and distributed programming languages, object-oriented techniques, and software engineering on the other hand. A considerable number of open issues has been raised, many of which are adressed in the DOCASE project and in the various other projects presented at the workshop. We would like to see a common understanding about the major issues, and about approaches and steps for resolving them.

5 References

[ALM85] G. Almes, A. Black, E. Lazowska, J. Noe
The Eden System: A Technical Review
IEEE Trans. on Software Engineering, Jan. 1985

[AND83] Andrews, G.R., Schneider, F.B.
Concepts and Notations for Concurrent Programming
Computing Surveys, Vol. 15, No. 1, March 1983, pp. 3 - 43

[AND82] Andrews, R.A.:
The Distributed Programming Language SR - Mechanisms, Design, and Implementation
Siftware - Practice and Experience, Vol. 12, 1982, pp. 719-753

[ARA87] Arapis, C., Kappel, G.
An Object Software Base
in: Tsichritzis, D.: Active Object Environments, University of Geneva, 6/1988, pp. 32 - 50

[BAU82] Bauer, F.L.
From specifications to machine code: program construction through formal reasoning
Proc. 6th Intl. Conf. SW Engineering, Tokyo 1982, pp. 84-91

[BEN87] Bennett, J.K.
The Design and Implementation of Distributed Smalltalk
OOPSLA '87 Proceedings, ACM 1987

[BIR84] Birrell, A.D., Nelson, B.J.
Implementing Remote Procedure Calls
ACM Transactions on Computer Systems, Feb. 1984

[BLA86] Black, A., Hutchinson, N., Jul, E., Levy, H.
Object Structure in the Emerald System
OOPSLA '86 Proceedings, ACM 1986

[BLA87] Black, A., Hutchinson, N., Jul, E., Levy, H., Carter, L.
Distribution and Abstract Types in Emerald
IEEE Trans. on Software Engineering, Jan. 1987

[BOO86] Booch, G.
Object Oriented Development
IEEE Trans. on Software Engineering, Feb. 1986

[BOR88] Borras, P., Clément, D.
CENTAUR: the system
ACM SIGSOFT/SIGPLAN conf. Practical SW Engineering Environments, Boston, MA, November 1988, pp. 14 - 24

[BRI78] Brinch Hansen, P.
Distributed Processes: A Concurrent Programming Concept
CACM, Vol 21, No. 11, 1978, pp. 934 - 941

[CAA87] Cartwell, J., Alderson, A.
The Eclipse two-tier database interface
Proc. 1st Europ. SW Engineering Conf., Strasbourg, F, 9/87, (Springer), pp. 129-137

[CAL87] Calton, P., Jerre, D.N.
Design and Implementation of Nested Transactions in EDEN
Proc. 6th Symp. on Reliability in Distributed Software and Database Systems, 1987

[COO80] Cook, R.P.:
*MOD - A Language for Distributed Programming
IEEE Trans. Software Engineering, Vol. 6, No. 6, 1980, pp. 563-571

[COX86] Cox, B.J.
Object Oriented Programming
Addison-Wesley 1986

[CUL87] McCullough, P.L.
Transparent Forwarding: First Steps
OOPSLA '87 Proceedings, ACM 1987

[DEC86] Decouchant, D.
Design of a Distributed Object Manager for the Smalltalk-80 System
OOPSLA '86 Proceedings, ACM 1986

[EBE88] Eberle, H., Geihs, K., Schill, A., Schoener, H., Scmutz, H.
Generic Support for Distributed Processing in Heterogeneous Networks
HECTOR Proceedings, Vol. 2, Springer 1988

[ELR82] Elrad, T., Francez, N.
Decomposition of Distributed Programs into Communication Closed Layers
Science of Computer Programming, VOl. 2, No. 2, 1982, pp. 155-173

[ERI82] Ericson, L..
DPL-82: A Language for Distributed Processing
Prof. IEEE 3rd Intl. Conf Distr. Comp. Systems, Ft. Lauderdale, FL, 10/1982, pp. 526-531

[FAG88] Fagerström., J.
Design and Test of Distributed Applications
Research Report, No. LiTH-IDA-R-88-22, University of Linköping, Sweden

[FAL89] Falk H.
Software vendors serve up varied palette for CASE users
Computer Design, Jan. 1, 1989, pp. 70 - 80

[FEL79] Feldmann, J.A.
High Level Programming for Distributed Computing
CACM Vol. 22 No. 6, 1979, pp. 353-368

[FOR86] Forman, I.R.
On the Design of Large Distributed Systems
Proc. 1st Intl. Conf. Computer Languages, Miami, Oct. 1986

[FRA85] Francez, N., Hailpern, B.
Script: A Communication Abstraction Mechanism
Operating Systems Review, No. 2., April 1985, pp. 53-67

[GAL87] Gallo, F.
The PCTE initiative: toward a European approach to software engineering
Proc. CRAI workshop on Software Factories & Ada, Capri, 5/1986 (Springer), pp. 16-29

[IEE88] Special Issue on CASE
IEEE Software, March 1988

[JUL88] Jul, E., Levy, H., Hutchinson, N., Black, A.
Fine-Grained Mobility in the Emerald System
ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 109-133

[KES81] Kessels, J.L.W.
Soma: A Programming Construct for Distributed Processing
IEEE Trans. on Software Engineering, Vol. 7, No.5, 1981, pp. 502 - 509

[KRA83] Kramer, J., et al.
CONIC: an integrated approach to distributed computer control systems
IEEE PROC., Vol. 130, No. 1, January 1983

[LEB82] LeBlanc, R.J., Maccabe, A.B.
The Design of a Programming Language Based on Connectivity Networks
Proc. IEEE 3rd Intl. Conf. Dist. Comp. Syst., Ft. Lauderdale, FL. 10/1982, pp. 532-541

[LIS82] Liskov, B.
On Linguistic Support for Distributed Programs
IEEE Trans. on Software Engineering, Vol. 8, No. 3, May 1982

[MŪH88] Mūhlhäuser, M.
Software Engineering for Distributed Applications: The DESIGN project
Proc. IEEE 10th Intl. Conf. on Software Engineering, Singapore, April 1988

[MSH88] Mūhlhäuser, M., Schill, A., Heuser, L.
Software Engineering for Distributed Applications: An Object-Oriented Approach
Proc. Intl. Workshop SW Engineering & Its Applications, Toulouse, F, Dec. 1988, pp. 264-284

[PEN86] Penedo, M.
Prototyping a Project Master Data Base for Software Engineering Environments
Proc. ACM Conf. Practical SW Eng. Environments, Palo Alto, CA, Dec. 1986, pp.1-34

[REN82] Rentsch, T.:
Object-Oriented Programming
ACM SIGPLAN Notices, vol. 17, no. 9, Sept. 1982

[RUM87] Rumbaugh, J.:
Relations as Semantic Constructs in an Object-Oriented Language
Proc. OOPSLA 87 in SIGPLAN Notices, Vol. 22, No. 12, Dec. 87, pp. 466-481

[SCH84] Scheffer, P.
Evolution towards a comprehensive software development environment
IEEE Proc. Compcon, Fall 1984, Arlington, Virginia, pp. 306-309

[SCH84] Scheiffler, R., Gettys, J.
The X Window System
ACM Trans. on Graphics, Vol. 5 No. 2, April 1987, pp. 79-109

[STA88] Staroste, R., Schmutz, H., Wasmund, M., Schill, A., Stoll, W.
A Portability Environment for Communication Software
HECTOR Proceedings, Vol. 2, Springer 1988

[TAN88] Tanenbaum, A.S., van Renesse, R.

A Critique of the Remote Procedure Call Paradigm
Proc. Research into Networks and Dist. Applications, Wien, April 1988. North Holland

[TSI88] Tsichritzis, D., Nierstrasz, O.:
Fitting Round Objects Into Square Databases
in: Tsichritzis, D.: Active Object Environments, University of Geneva, 6/1988, pp. 202-218

[WAS86] Wasserman, A., Pircher, P.
A Graphical, Extensible Integrated Environment for Software Development
Proc. ACM Conf. Practical SW Eng. Environments, Palo Alto, CA, Dec. 1986, pp.131-141

[WEG87] Wegner, P.
Dimensions of Object-Based Language Design
OOPSLA '87 Proceedings, ACM 1987

[YOK87] Yokote, Y., Tokoro, M.
Experience and Evolution of Concurrent Smalltalk
OOPSLA '87 Proceedings, ACM 1987