VIEWPOINT TRANSFORMATION

Kazi Farooqui, Luigi Logrippo
Department of Computer Science,
University of Ottawa, Ottawa K1N6N5
Canada.
Internet: farooqui@csi.uottawa.ca, luigi@csi.uottawa.ca

ABSTRACT

The ODP Systems are specified from five abstractions: Enterprise Viewpoint, Information Viewpoint, Computational Viewpoint, Engineering Viewpoint, and Technology Viewpoint. The major gap in the ODP Reference Model is felt to be in the area of "viewpoints"; in particular the question of how to ensure consistency between descriptions of the same system in different viewpoints remains open, as a challenge to researchers in this area. In this paper we discuss a *viewpoint transformation* approach to ODP system design which ensures consistency between computational and engineering viewpoints. The viewpoint transformation process is considered in the framework of a formal design process, starting from an abstract description of the distributed application (computational specification) and ending with the specification of the actual realised system (engineering specification). One possible approach: the *correctness preserving transformation* supported by FDT LOTOS is explored for achieving computational to engineering viewpoint transformation.

1.0 INTRODUCTION:

The purpose of the Open Distributed Processing (ODP) framework of abstraction is to partition the concerns to be addressed in the design of distributed systems. To deal with the complexity of distributed systems, the framework of abstractions considers the system from a set of interrelated viewpoints, where each viewpoint represents a different abstraction of the original system. A viewpoint leads to a representation of the system with emphasis on specific set of concerns, and the resulting representation is an abstraction of the system, that is, a description which recognizes some distinctions (those relevant to the concern) and ignores others (those not relevant to the concern). Different viewpoints address different concerns, but there will be a common ground between them. The viewpoints must treat this common ground consistently, in order to relate viewpoint models and to make it possible to assert correspondences between the representations of the system in different viewpoints [1]. The ODP viewpoints can be used to structure the specification of a distributed system, and can be related to a design methodology. Design of the system can be regarded as a process that may be subdivided into phases related to different viewpoints. In this paper we explore *viewpoint transformation* approach to ODP system design which ensures consistency between computational and engineering viewpoints.

This paper is organized as follows: Section 2 identifies the relationship or the mapping between computational and engineering viewpoints. The concept of viewpoint transformation along with the issue of language support for attempting the viewpoint transformation is presented in Section 3. Interface is an important architectural concept in ODP. Computational interface template [5] forms the basis of viewpoint transformation. Section 4 discusses the relevance of the interface concept in ODP and the rationale behind formal specification of interfaces. The issues in modelling the computational interface template in LOTOS and possible solutions are outlined in section 5. A simplistic mapping between computational and engineering viewpoint is illustrated in section 6. In section 7 we explore the applicability of LOTOS c*orrectness preserving transformations* (CPT) techniques [3] for achieving computational to engineering viewpoint transformation. Section 8 lists some benefits of viewpoint transformation approach to ODP system design. Directions for future work are mentioned in section 9. Conclusions are drawn in section 10.

Quite clearly this is an area of future research and ideas are presented to provide motivation and discussion.

2.0 Relationship between Computational and Engineering Viewpoint:

This section identifies the relationship that exist between ODP Computational and Engineering model. The different aspects of Computational and Engineering model are explored. By identifying the correspondence between computational and engineering models, the true *nature of transformations* that could be applicable become apparent.

These two models play a key role in an architectural approach, as they allow to express some fundamental properties of the ODP architecture. Both models are trade-offs between different types of requirements and various solutions that may be envisaged.

To be able to benefit from these models, it is necessary to establish a firm link between them, and also between the models and the real world. This means for instance being able to map computational concepts onto the engineering viewpoint, or to identify engineering modules in a real world implementation. These links play a critical part in the architectural approach to system design. The interest of the computational model is directly related to the existence of a mapping enabling it to relate it to engineering concerns [9].

The ODP computational model is fully generic and can be applied across a wide range of application domains (from office information systems to computer integrated manufacturing and telecoms), and to any type of distributed application (in particular, with or without real-time requirements) [6]. The computational viewpoint provides a *service-oriented* view of the system.

The ODP engineering model provides an infrastructure or a distributed platform for the support of the computational model. The engineering model also provides generic services and mechanisms capable of supporting distributed applications specified in the computational model. The engineering viewpoint is centered around the ways the application may be *engineered* onto the system.

The computation model is object based: applications are collections of interacting objects. In this model, objects are the units of distribution, encapsulation, and failure. Interaction between objects takes the form of operations at (named) interfaces.The only way to access the state of an object is to invoke some operation on an interface of the object.

The engineering model is also object based: the set of basic services, identified in this model, can be viewed as a collection of interacting objects which together provide support for the *realization* of interactions between distributed application components.

The computation model describes the coarse-grained structure of an application, i.e., the application components and their interaction at an abstract, system independent level. Each coarse-grained entity of a distributed application is represented by an object, called *computational object*, with a (set of) well defined interface(s), called *computational interface*. The computational modelling activity comprises the specification of computational interfaces and the application-level communication between the (interfaces of) computational objects, referred to as *computational interactions*. Remote (computational) interactions are expressed at a high level in terms of application-level operations rather than in terms of physical messages.

For application-level processing, a distributed object based engineering environment is offered. It supports location independent object invocations and object mobility and provides high level of distribution transparency[1]. The engineering model is concerned with mapping of an application to a concrete distributed system. Concrete application configurations and support infrastructure issues such as component placement, distribution, and performance are addressed here. The application-driven issues of configuring transparency mechanisms and communication (protocol) objects are relevant here. The selection of transparency and protocol objects, among many other support mechanisms, tailored to application needs, forms an important task [7].

The computation model can be viewed as a (language, operating system, and machine-independent) framework for structuring and designing distributed applications, and identifying the interactions between distributed application components. The computation model represents a distributed system as seen by application designers and programmers. From this viewpoint, an ODP system appears as a (large) programming support environment capable of building and executing distributed applications. *This is equivalent to the specification of an abstract machine, whose (abstract) realization is the purpose of an engineering model* [8]. The engineering model provides a machine-independent execution environment for distributed applications. The computational model hides from the application programmer details of the realization of the underlying abstract machine that supports it.

The ODP computational model defines the programming features and system components that should be available to distributed application programmers. Maximum engineering flexibility is obtained if all computational statements are expressed declaratively, i.e., state *what* is required (computation), not *how* it is to be provided (engineering). *This permits the application of tools to automatically generate the engineering support code necessary to meet specific requirements (computational) in specific environments (engineering).* Such tools perform a mapping from the 'idealistic' computational requirements,i.e., the world as seen by application programmers, onto a 'realistic' engineering solution, i.e., a world as seen by operating system/communication system designers [10].

The computational model provides application designers/programmers with a generic set of tools for building distributed applications, independent of the underlying engineering issues. Hence, distributed applications constructed using languages and

1. The distribution transparencies proposed for ODP Engineering Model are: access transparency, location transparency, migration transparency, concurrency (transaction) transparency, replication transparency, failure transparency, and resource transparency.

environments conforming to the ODP computational model will be portable to other environments that also support this model [10]. It is the job of the engineering model to prescribe solutions to the requirements asked in the computational model.

In ODP, the notion of computational model provides the equivalent of a programming language environment, for use on top of an abstract machine realized by an ODP engineering infrastructure [8]. Such a computational model should contain programming language features commonly found in advanced object-based distributed platforms. As such, the computational model (viewed as an application programming environment) could be defined as:

1. An abstract language for building distributed applications.
2. A library of system calls.
3. An abstract machine. This being the most general.

To distribute applications, the engineering model specifies how separated application components can interact and communicate with each other, providing transparency guarantees offered in the application programming environment. The mechanisms of the engineering infrastructure are logically hidden from the operation invocation abstractions of the computational model.

The engineering model identifies the functionality of the basic system components that must be present, in some form or other, in order to support the computational environment described in the computational view. Hypothetically, there may be several engineering models for a particular computational environment, reflecting the use of different system components and mechanisms to achieve the same end. The issue in the computational viewpoint is *what* (objects, interfaces, interactions, environment constraints); the engineering viewpoint prescribes the solution as to *how* to realize these computational objects and interactions such as to satisfy their environment constraints.

The computational viewpoint is primarily intended for providing the concepts needed to explain how services can be programmed in a form suitable for distribution. The computational model may thus be characterized as focussing on the organization of applications in architecturally conformant ways rather than on mechanisms used to distribute, or more generally, support them in the system (which is visible in the engineering viewpoint) [9]. This model should thus provide the application designer with tools that facilitate in observing what support is available from engineering infrastructure. The computational model should allow to express in conformant ways architectural features such as *interaction* and *sharing* of common sub-applications. Since the computational model is object-oriented, it is quite convenient to express these types of relations: for instance, inheritance trees allow to express reuse of components between applications; message-passing helps to enforce interaction models able to express for instance conformance in infrastructure usage as well as inter-application interfaces; similarly the encapsulation principle is very useful to allow a smooth coexistence of several applications in a system.

In order to build portable applications that can run on several ODP engineering platforms, an "implementation specification" of the application described in terms of the computational model must be available. If a more abstract specification of the application exists, then a refinement mapping between both specifications must be available. More generally, one can envisage to follow, for ODP applications, a design approach combining top-down refinement and reusability of components [8]. The support of a design trajectory as suggested in [15] is thus needed for ODP application development.

From a distributed software engineering point of view, the computation and engineering viewpoints are most important; they reflect the software structure of an application most closely and constitute an ODP support environment.

3.0 VIEWPOINT TRANSFORMATION:

At the heart of the separation between ODP computational and engineering model is the *idea of a tool-driven transformation between the abstract computational description of a distributed application and its mechanization in terms of the engineering model*.The engineering model *animates* the computational model [11].

In passing from the computational viewpoint to engineering viewpoint, concerns shifts from the specification of computational structures(e.g. computational objects, computational interface templates, etc.) and statements of necessary properties of interactions between object interfaces (e.g. transparency requirements) to engineering mechanisms capable of *realizing* these properties [12].

3.1 Motivation:

The computational viewpoint is the starting point of a distributed application design. The division of a distributed application into computational objects is based on the distribution properties of the application.Once the distribution of application function is done, it results in a set of *computational objects*; these objects may be specified in some formal languages. This specification constitutes *computational object template*. A *computational interface* is a projection of computational object's behavior, seen only in terms of a specified set of observable actions. The *computational interface template* is a specification of named operations (with their arguments) supported at the interface together with the ordering and concurrency constraints between operations(behavior). The operations are qualified by QOS/transparency attributes (or constraints).

The computational view is that of an "object world" populated by a set of concurrent interacting objects which may support multiple interfaces as service provision points.The engineering model provides an infrastructure (a virtual machine) for the support of distribution transparent interactions at the interfaces of computational objects. Computational environments must have:

1.Language support for building/executing applications.

2.Language compilers for transforming *computational level interactions* into *engineering support mechanisms*.

Computational objects represent distributed application components which interact with one another in a *distribution transparent* abstraction.Whereas the focus of the computational model is on (computational) object *interfaces* and the *interactions* that occur at these interfaces, the engineering model provides the *structures* which realize these computational interactions.

Viewpoint transformation is a mapping from an abstract computational view of "object interactions" to a realization of an engineering solution comprising of basic engineering objects (BEOs), transparency objects, stubs, nucleus (communication support) objects, etc. Service and transparency requirements specified at a high level in the computational model have to be mapped to adequate engineering subsystem services. The type of service required from the engineering subsystem may be specified explicitly or be derived from the application interaction structure, and should be mapped, as far as possible, automatically to required engineering subsystem functionalities.

3.2 Formal Methods in ODP Viewpoints:

FDTs are a meta-technology that is not directly usable within products like ODP systems. Instead this technology is used for the specification and design of parts which in turn can be used in products. The application of FDTs in the formal design trajectory of open distributed systems has been proposed earlier in [8], [15] and [29].

The issue is what language is to be used for describing computational view of a distributed application. It should be programming language independent and capable of expressing computational structures and interactions in a system-independent abstraction. Formal languages like LOTOS [4] seem to be likely candidates for this purpose.The benefit of using such languages is that they are based on formal mathematical semantics and hence amenable to correctness preserving transformations and conformance checking (the derived engineering solution conforms to more abstract computational specification).

We view computational viewpoint as an "abstract specification of interactions, at interfaces, of distributed objects of a distributed application". And engineering viewpoint as a "realization of computational interactions in terms of engineering structures (such as BEOs, composition of transparency objects, nucleus etc.).

Furthermore, we view a computational specification as akin to *service specification* [13] (in OSI terms) and engineering structures akin to p*rotocol entities* [13] (in OSI terms) which together support (and must conform to the service specified in) the computational specification.

We are planning to specify computational interface template (and hence computational interactions) in LOTOS. And we intend to transform the computational specification into a less abstract realization of engineering structures (in engineering viewpoint). At present we are approaching this kind of "viewpoint transformation" using *correctness preserving transformations [3], decomposition of functionality* [14], incremental specification [16], and techniques similar to those used for transforming a service specification into the specification of protocol entities [17], [18].

However, in the more traditional service to protocol transformations, called *protocol synthesis*, there are usually two protocol entities synthesized from a given service specification. Whereas a computational specification of (distributed) object interactions is realized in the engineering model by (much) more than two engineering structures (e.g., BEOs one for each computational interface, a number of transparency and stub objects etc.). It appears that the synthesis of engineering model from the computational specification is a generalization of protocol synthesis problem and techniques more general than those proposed for protocol synthesis may be necessary.

3.3 Starting with Interface Templates:

A computational interface template identifies all the operations supported at the interface together with their application-level interaction semantics. The information pertinent to the *transformation* exercise, such as distribution transparency, quality of service, and other environment attributes [5] associated with the operations and their ordering and concurrency constraints, is available in an interface template. Furthermore an interface template is a specification of projection of object's behavior (seen only in terms of a specified set of observable actions).

We think the computational interface template would be the starting point to get an engineering solution from an abstract computational specification. A computational specification is a composition[2] of concurrent computational interfaces supported by computational objects which represent distributed application components.

Computational interfaces can support distribution, concurrency, and synchronization. Interface specifications can be used for the generation of stubs that implement distribution transparency, concurrency and synchronization aspects of the computational model.

An important issue is how to specify the constraints/attributes, e.g., transparency, QOS, and environment constraints, of operations and their ordering and concurrency constraints in LOTOS, such as to make the specification more amenable to automated transformation. The criteria that should apply to decide what to put in and what ought to be left aside in computational interface template are: efficient compilation, i.e., automatic transformation into infrastructure objects and efficient static type checking.

3.4 What is to be transformed:

In the computational model the issue is how to specify a distributed application in terms of interacting application components, their interfaces, transparency and communication requirements of computational operations, etc. and the issue in the engineering model is how to *realize* the computational specification.

The constraints/attributes in the computational specification of a distributed system that are to be transformed into engineering objects which support these attributes are:

1. Distribution transparency attributes.
2. Quality of service attributes (both service and communication QOS).
3. Ordering and concurrency constraints.
4. Other environment attributes (from enterprise and information viewpoints).

In an interface type description, a large set of QOS constraints can be specified: volume, cost, quality of perception, criticality, dependability, survivability, security, etc. There is a priori no bound on what quality of service parameters could be explicitly included in interface type descriptions. Even behavioral information is possible (some is already in).

The following activities are involved in viewpoint transformation:

1. The modelling of interfaces and interactions (at the interfaces) of the computational objects. This constitutes a high level computational specification of distributed application interaction semantics.
2. Derivation of the transparency, QOS, and other environment attributes/constraints from the computational interface specification. Each of the operations included in the interface template may specify its own transparency requirement and other environment constraints. The engineering infrastructure must provide the composition of corresponding transparency and support mechanisms when the operation is invoked.
3. A series of transformation steps resulting in the identification and specification of engineering infrastructure modules to support the application interaction specified in the computational viewpoint. This constitutes the engineering solution.

It involves techniques for describing the dependency of objects at one level of abstraction on those at another level.

2. This composition is expressed by the rich set of LOTOS composition operators.

3.5 Consistency Constraints: As specified in [5], an engineering specification S_2 is consistent with a computational specification S_1 if

1. S_1 is a configuration of sub-objects, or a single object, and
2. S_1 can be transformed into a configuration of basic engineering objects, a set of stub objects, binding objects, protocol objects and a nucleus object, or a set of replicated such configurations, S_1', such that
3. S1′ is behaviorally compatible with S_2, and
4. each interface of S_1 corresponds to an interface in S_1', and
5. in case of a set of replicated configurations, the basic engineering objects are configured as replica groups.

4.0 INTERFACE CONCEPT IN ODP:

ODP is a fairly typical distributed computing system, consisting of a number of interacting components which form a complex web of interaction and dependency. ODP objects/components are described in terms of their behavior and information exchange at their interfaces.

An interface description of an object describes how the environment can interact with the object and vice versa. An interface is a first class entity in ODP, that consists of a set of operations and of behavioral and environmental constraints on their invocation.

The objective of ODP is to enable distributed system components to interwork seamlessly, despite heterogeneity in equipment, operating systems, networks, languages, data base models or management authorities.

In order to achieve these ambitious goals, the ODP must accomplish two things[19]:

1. An ODP system must supply the distribution transparency mechanisms to mask the underlying heterogeneity from users and applications.The realized set of components that provide these transparencies (together with the nucleus and communication support) constitute the ODP engineering infrastructure. Distributed application components will then inter-operate through ODP engineering infrastructure. The ODP infrastructure is the platform that will make the *network computing* a reality.
2. ODP must provide a technique for the specification of interfaces.The ODP infrastructure will allow client application components to access the server no matter:
   a. where the clients and server are located in the network.
   b. what programming languages were used for the clients or server.
   c. what local operating systems are involved.

4.1 Basic ODP concept:

The objective of ODP is inextricably related to the problem of interface definition. The components of a distributed system might be developed in different environments using different technology. It is therefore essential for the developers of a client component to have a precise specification of server's interface; the specification must be unambiguous and implementation independent. At run-time, the interface specification is the vehicle for ensuring that the interface expected by the client is compatible with the one offered by the server, so that the infrastructure can effect a type-checked binding [19].

An interface description, called *interface template*, describes an ODP object's role as a *server*, but in addition the interface description may also describe the object's role as a *client*.

In general, interfaces are boundaries between architectural elements which have

been identified as being of some significance for specification or design purposes. Each architectural element in the system is considered as an *object*. Basic building blocks at the computational level are:
1.Objects, and
2.(Abstract) Interfaces.

Objects are the units of structure. They encapsulate functions and data which are only accessible through well defined interfaces. A distributed application is then seen as a collection of *interacting* and *synchronizing* objects; interaction and synchronization taking place at interfaces between objects [20].

An interface has a *type*, which characterizes the interactions that can take place at this interface. An object can have several interfaces, which need not be of the same type. In a system, an interface can be provided by several objects.

The distinction between object and interface is the distinction between the function-provider and the function. It is the core of the client-server approach, where certain objects (servers) are responsible for offering certain functions to others (clients). Servers are responsible for providing certain interfaces; clients are responsible for invoking functions that are accessible through the offered interfaces [20].

4.2 Formal Interface Specification: Why

In the design of distributed applications, it is often necessary to put together parts written in different languages and developed independently. A language-independent specification of component interfaces makes this possible: language-independent specification helps ensure that representation of the interface in different languages have consistent semantics. The use of more than one programming language is unavoidable. The existing software components can be encapsulated inside new interfaces.

Formal languages can be used as the ODP computational languages. It should be possible to define interface types and behavior in a manner independent of actual localization of interfaces and of programming languages.

The interface specification can be used as an input to the transformation tools for the generation of engineering level code.

5.0 MODELLING COMPUTATIONAL INTERFACES IN LOTOS:

The problem of interface definition is central to ODP. In the ODP computational model, the interactions at the interfaces of computational objects are specified in terms of operational and non-operational interfaces. The specification of non-operational interfaces does not include the statement of a set of operations which can be performed at the interfaces. This ability to omit the operations is included to allow the specification of continuous media such as audio or video. This section describes the modelling of operational interfaces.

In LOTOS we can model an interface as a gate at which a number of constraints act to impose allowable signature/behavior and information content on all events occurring at that interface. A LOTOS event then denotes an interaction of some nature (e.g. communication of message) between the object and its environment. The architecture of interacting objects may then be captured in LOTOS by combining the involved objects' interface description in an appropriate parallel composition.

The specification of computational interface template comprises:
1. Operation specification

2. Property specification
3. Behavior specification
4. Role indication

5.1 Operation specification: The definition of operations that this interface supports. Operation specification includes:
a. Operation name: Each operation has a local name within an interface template.
Data Specification:
b. The number, sequence, and type of arguments that may be passed in each operation.
c. The number, sequence, and type of results that may be returned from each operation invocation.

This specification can be done using LOTOS ADT expressions. This constitutes the *operation signature*. Both operation names and arguments can be represented as abstract data types. Operation invocations are event (or experiment) offers on gates which model computational interfaces.

5.2 Property specification: The property specification in the computational interface template defines the following:
a. distribution transparency requirement on operation invocation.
b. quality of service (including communication quality of service) attributes associated with the operations.
c. other environment constraints (e.g., those arising from enterprise and information viewpoints) on operations.

These attributes may be associated with individual operations or the entire interface. Property specification in the computational interface template has direct relationship to the realised engineering structures and mechanisms.

Property specification constitutes an important component of computational interface template. There seems to be number of approaches for specifying distribution transparency, QOS, and other environment attributes in LOTOS. Transparency attributes and quality of service requirements may be represented in LOTOS as another abstract data types (like operation arguments) or they may be specified as guards or selection predicates (constraints) so that appropriate transparency objects and other support mechanisms can be obtained as a result of transformation process. This problem is both important and difficult. Currently, there is a lack of clear understanding about the property specification in the computational interface template. We plan to concentrate on this issue.

5.3 Behavior specification: Defines the behavior exhibited at the interface[3]. All possible ordering of operation invocations at or from this interface are specified. This includes ordering and concurrency constraints between operations as well as sequential and disabling (interrupt) type operations. All these constraints can be specified using a combination of rich set of LOTOS composition operators[4]. In particular the following generic interface characteristics can be specified in LOTOS:
1.Usually interactions occurring at an interface take the form of *request-confirm* pair, in an

3. In the current RM-ODP standard, interface type definitions do not adequately cover the notion of behavior.
4. This includes sequential, parallel, enabling, and disabling composition operators.

asynchronous communication model. These are referred to as *interrogation* type operations in ODP.

2. An interface (service-provider) can support a number of concurrent (request-confirm type) interactions.
3. Often the previous history of interactions at the object interface affects information content and behavior of all subsequent interactions at that interface. This can be achieved by synchronising every invocation by history constraints (state information).
4. Execution of an operation may be interrupted (or disabled) by the invocation of another operation.
5. In the case of transaction (or ACID) operations, there is a need to specify conflict and commutativity rules which are elements of behavioral specification.

Specification of ordering and concurrency constraints using path expressions has been done in ANSA [21].

An important issue is to explore if *behavior* has any impact on the configuration of transparency support objects in the engineering model.

5.4 Role indication: Often an object assumes the roles of either *client* (invoking services encapsulated by other objects) or *server* (providing services to other objects). All interactions of an object, both as a client and as a server, between it and its environment[5] occur at object interfaces. Until now, an interface is represented by a single LOTOS gate, but it is sometimes convenient to partition the complete interface of an object into a number of more limited interfaces (also represented as LOTOS gates). This allows us to explicitly partition server role interaction concerns from client role interaction concerns, and to explicitly reflect compositional architecture in terms of interactions with other objects. If gates cannot be used, roles can be represented by constants exchanged in interactions.

6.ILLUSTRATION:

This section illustrates a simplistic mapping from computational to engineering view of object interactions. In passing from computational viewpoint to the engineering viewpoint, concerns shifts from the specification computational structures and statements of necessary *properties* of interactions[6] between objects to engineering mechanisms capable of ensuring these properties [12].

The computational model defines the semantics of computation in terms of interactions between computational objects. The engineering model specifies the structures for implementing the abstract computational model (for e.g., mapping of interactions onto local versus remote calls, mapping of requirements onto support mechanisms, etc.).

As shown in figure 1, what is required is a *refinement* or *decomposition* of computational interactions (at the interfaces of computational objects) into a composition of transparency and communication support mechanisms which regulate and enable distribution in the engineering model. The computational model contains few objects but ascribes properties to their interactions; while in the engineering model the issue is how to realize the computational interactions by means of supporting engineering objects such as to construct the required properties by their interaction.

The synthesis of engineering model implies the introduction of several engineering

5. A computational environment is a population of interacting computational objects.
6. for e.g., distribution transparency requirements, specific communication requirements, etc.

objects and interfaces through which the computational interfaces interact.

Quality of service requirements, distribution transparency requirements, and other environment constraints are specified individually for each operation or for the entire interface in the computational interface template. What is required is a translation of constraints in interface type description to insertion of appropriate mechanisms in the engineering model.

Basic engineering objects (BEOs), shown in figure 1, are the run time representations of corresponding computational objects. A BEO is a corresponding computational object enriched with extra state and interfaces (operations) to enable it to cooperate with the transparency stack and protocols.

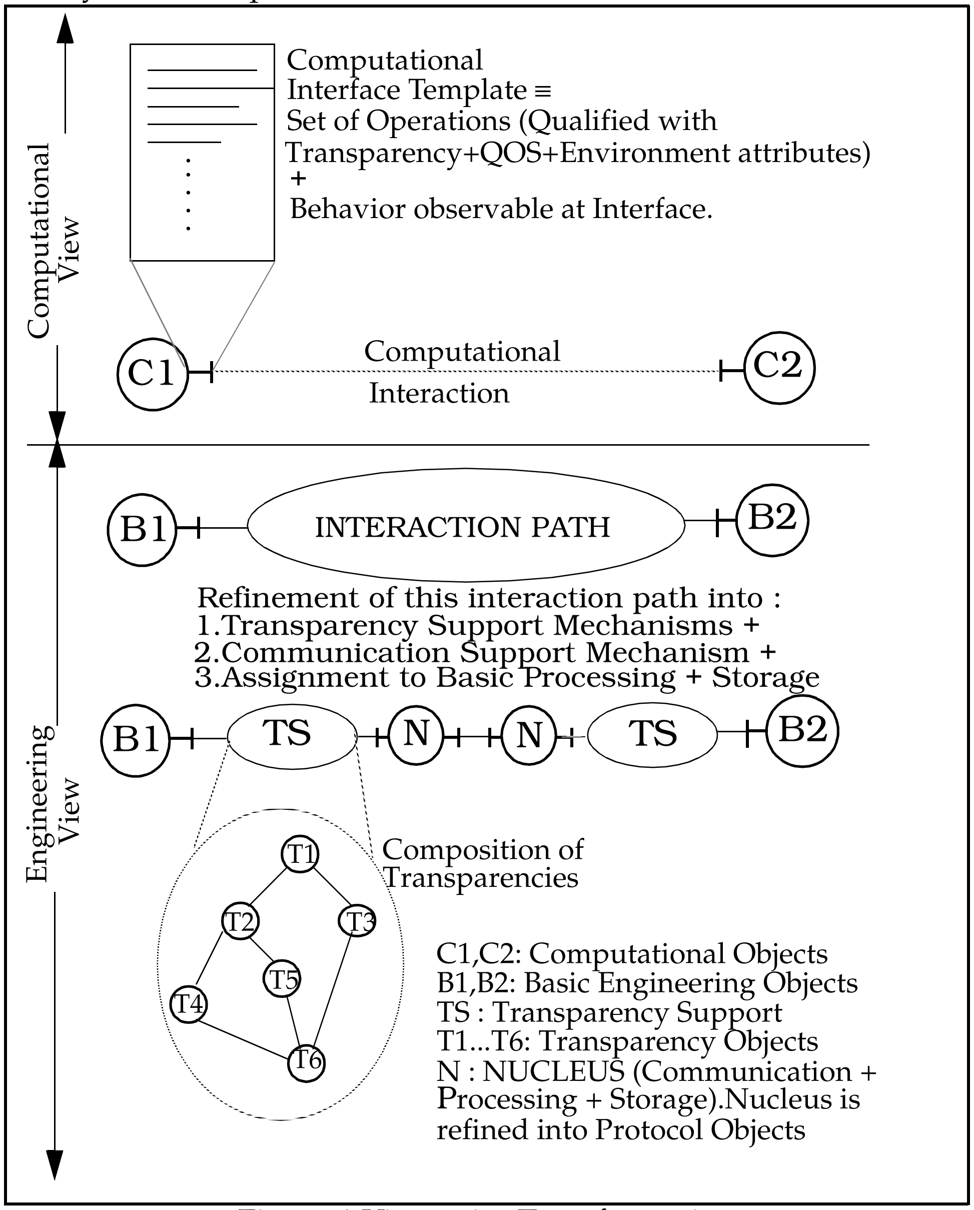
Computational View
Computational
Interface Template ≡
Set of Operations (Qualified with
Transparency+QOS+Environment attributes)
+
Behavior observable at Interface.
C1
Computational
Interaction
C2
Engineering View
B1
INTERACTION PATH
B2
Refinement of this interaction path into :
1.Transparency Support Mechanisms +
2.Communication Support Mechanism +
3.Assignment to Basic Processing + Storage
B1
TS
N
N
TS
B2
T1
T2
T3
T4
T5
T6
Composition of
Transparencies
C1,C2: Computational Objects
B1,B2: Basic Engineering Objects
TS : Transparency Support
T1...T6: Transparency Objects
N : NUCLEUS (Communication + Processing + Storage).Nucleus is refined into Protocol Objects

Figure 1 Viewpoint Transformation

An important issue is deciding whether a computational object can be decomposed into multiple engineering objects, or a group of computational objects can be mapped onto a single engineering object. A computational object with multiple interfaces could be decomposed into several engineering objects. The only constraint is that interfaces should be preserved in the transformation. A computational interface becomes a single (but perhaps replicated) engineering interface. Objects can be split (or merged) but not the interface.

Computationally, an object's environment is everything the object can interact with. In the engineering model the environment can be considered as being composed of two parts:

1.the "*remote*" basic engineering object(s) with which this object interacts (indirectly).
2.the *channels* between this object and the remote objects.

The channels themselves can be broken into three parts:

1.the "local" environment: the cluster/capsule/nucleus/node [5] in which this object is located (e.g. an object with certain security requirements cannot be placed in an unsecured node).
2.the "remote" environment of the interacting object.
3.the communication path between them.

So to satisfy the object' environmental constraints:

1.it must be placed in an appropriate local environment.
2.the remote interacting objects (or BEOs) must be selected which are both appropriate in themselves and located in an environment suitable for binding to appropriate channels.
3.adequate channels be constructed using communication objects.

Thus, environment attributes would appear to get mapped into:

1.local environment constraints.
2.remote object constraints.
3.remote environment constraints.
4.binding constraints.

7.0 CPT APPROACH TO VIEWPOINT TRANSFORMATION:

The viewpoint transformation process is considered in the framework of a formal design process[2], starting from the abstract description of the system (computational specification) and ending with the actual realised system (engineering specification). LOTOS allows the formalization of the design process, where the design step can be seen as a transformation of the specification produced in the previous step into a new, more refined, one. The designer must decide which transformations will be applied in each step (semi-automatic), so that initial requirements/constraints will be maintained during all the design process.

Correctness preserving transformations (CPT) [3] can help the designer along the design trajectory. These transformations preserve the correctness of each new refinement, as they maintain some equivalence relation with the previous refinement. This section explores the possibility of applying CPTs during viewpoint transformation process.

The existence of a computational model in ODP, taken as a basic implementation level necessary to ensure portability of applications, requires the formal support of a full design trajectory [8].

From an architectural point of view the issue is to define in broad terms the nature of transformations which can operate on the computational specification of distributed applications.

7.1 Structuring of computational specification: The first step in the computational modelling of distributed application is the structuring (or decomposition) of the application into application components, referred to as *computational objects*, identification of *interfaces* (interaction points between application components) and *interactions* that occur at these interfaces. Generally, functionality decomposition is a way to achieve militarization of design. Militarization serves several architectural purposes, like separation of concerns, identification of independent system components, reflecting the physical distribution of systems, etc.

The *splitting process* transformation [3] or *decomposition of functionality* [14] can be used to split a computational LOTOS specification of a distributed application into application components, called *computational objects*. The observable behavior of the resulting interacting processes is the same as the observable behavior of the original process. However as specified in [3], the input to this transformation must be in action prefix [4] form.

Once the *computational interfaces* have been identified, the interactions that occur between these interfaces must be specified. The *computational interaction* specification is then qualified with distribution transparency, QOS, and other environment attributes. It is this specification of computational interactions that forms the basis for viewpoint transformation.

7.2 Identification of Concurrency: The computational view is that of an object world populated by concurrent interacting objects. These objects can be obtained from the monolithic specification [22] of a distributed application by exploring the potential parallelism inherent in the specification. In the design of systems by step-wise refinement, it is highly desirable to have some means of decomposing a behavior into several parallel sub-behaviors. The *inverse expansion* transformation described in [23] helps the designer in this task. The main interest of this transformation is that it makes explicit the parallelism of the original computational description.

7.3 Functionality composition: The need for composing the *functionality* (components) based upon the offered interfaces arises in both computational and engineering modelling of distributed applications.

In the early steps of computational modelling, concurrent computational objects are used to capture the elements of functionality of the distributed application, possibly viewed as conceptually independent constraints on the behavior of the latter (constraint-oriented specification [22]), or components of an abstract architecture. At this stage of computational modelling the concurrent computational objects may be regrouped (i.e., the composition and synchronization relation between the objects may be altered) for exploring new representation of the specified functionality/architecture, that may turn out to be more natural, or logical for the distributed application under consideration. This kind of transformation may be attempted using the LOTOS *regrouping parallel process* transformation [3].

At the subsequent stage of engineering modelling, process regrouping may be used for letting the specification reflect the structure of a concrete architecture. Possibly, some constraints are imposed on the concrete (engineering) specification by the need to take into consideration pre-defined components (such as transparency support objects). Every computational interaction, may require the support of a specific set (and configuration) of transparency objects. With regrouping parallel process transformation, it is possible to specify the desired interconnection pattern between transparency objects, and hence to configure a standardized interconnection of transparency objects into the configuration required for the support of the given computational interaction.

Furthermore, regrouping can be used for achieving predefined interconnection patterns or for optimizing communication costs.

7.4 Interface Design: Where the designer has freedom to compose computational (or engineering) interfaces, it is meaningful to establish binding between interfaces based on certain criteria, or in order to achieve some optimization.

In the process of transformation from computational to engineering realisation of a distributed application, computational (or engineering) interfaces may be *split* or *merged*. There are different motivations for this transformation at different steps of design trajectory.

During computational modelling, architectural considerations may suggest the splitting of computational interfaces on the basis of its mapping into the interfaces of cooperating parts of the system (i.e., on the basis of *type* in terms of object-oriented paradigm); such a mapping could be necessary because in the computational modelling it is desired to refrain from the details of the communication (which lie hidden in the engineering viewpoint). In other words, in such a transformation step a meaning and structure is being attached to interaction points[7]. Also, separation of concerns criteria may induce that different communication constraints be represented by different interfaces in an abstract computational specification, whereas the communication itself will take place at a single communication channel (in the engineering infrastructure). On the contrary all the interfaces that have the same communication constraints or transparency requirements may be *merged* into a single interface. (This may be necessary because in the engineering realisation, computational objects may be split or merged but not the interface).

This kind of transformation, called *interaction point rearrangement* [3], is achievable in LOTOS, where the *gates* model the *interface* concept. An input LOTOS specification P consisting of a set of gates is transformed into another LOTOS specification Q consisting of a set of gates which is different from that of the input specification such that P and Q are behaviorally equivalent[8]. The gates of Q can be obtained from those of P in several ways. For instance, a certain gate g in P can be *split* into different gates $g_1,....,g_k$ in Q. Alternatively, several gates $h_1,.......,h_n$ of P can be *merged* or *integrated* into a single gate h of Q. For each event occurring in g_i in Q, there will be a uniquely determined event of P occurring in g, which can be recognized by interaction parameters. Also when an event occurs at gate h, it must be possible to uniquely find out the gate h_i of P at which the corresponding event should have occurred. Thus, no information in P is lost in Q.

At the later stages of design refinement, this transformation can be used in order to make a computational specification closer to target engineering model structures and objects. For instance, it can be used to split the interfaces of *engineering objects* (which are the run time representation of *computational objects*) so as to map onto the interfaces of transparency objects, stubs, nucleus objects.

In general gate splitting or merging can be applied in order to make sure that all values (data types) exchanged through different occurrences of a certain gate within a

7. The terms interaction point and interface are used interchangeably.

8. Bisimulation equivalent.

specification be of the same type. This will result in forcing *typed gates* thus making LOTOS modelling of computational and engineering issues of a distributed application closer to that of a typed programming language.

Finally, splitting a gate can facilitate the improvement of a specification in terms of increased parallelism. In fact, different actions which are necessarily sequential when performed on a single gate might be executed in parallel on different gates into which the former has been split.

7.5 Computational specification refinement: One of the important aspects of ODP design methodology should be the support of step wise refinement approach towards design. At the computational modelling stage of design trajectory, the designer abstracts from the details of the system. These details are incorporated later in the engineering modelling, when the proper design decisions are taken, and must be verified using a mathematical framework. Computational modelers should have the freedom to represent a set of possible realizations of a distributed application at a certain level of abstraction, so that the set can be restricted according to design decisions at the stage of engineering modelling, since engineering environments normally may not support non-determinism. This kind of support for step-wise refinement of design can be explored in LOTOS through *functionality extension* [24] and *resolution of non-determinism* [3] transformations. The *conformance*, *reduction*, and *extension* relations, well-established in LOTOS theory, provide the basis for this type of reasoning.

Another instance of the use of *resolution of non-determinism* transformation is the following: In the computational modelling sometimes it may be desired to describe minimal conditions for interactions to occur, leaving some implementation freedom for specific event parameter values. Such conditions can be represented in LOTOS as relations (occurring in guards), involving the event parameters and possibly some (internal) state values. The use of relations allows a multitude of parameters to be valid ones, which characterizes non-determinism. The non-determinism introduced by the use of relations can be removed by proper replacement of these relations by functions, which reduces the number of valid parameter values to single one.

7.6 Communication context refinement: Interactions at the interfaces of computational objects may be represented as multi-way synchronization. Multi-way communication may imply an agreement between computational objects on the values to be passed or it may be used for the purpose of synchronizing all the constituent computational objects involved in the distributed communication. Additionally, multi-way synchronization of computational interfaces may also mean performing an atomic action. However, the engineering environment may not have the capability to perform multi-way synchronization within a single communication construct; only two way communication may be realizable.

This kind of transformation can be attempted in LOTOS using *multi-way-to-two way synchronization*. A process *P*, in which an action on a gate *a* can be simultaneously performed by more than two subprocesses, is transformed into an equivalent process *Q* in which each action on a gate *a* is performed by at most two subprocesses.

7.7 Distribution transparency transformation: In a LOTOS specification of the computational interface template, the distribution transparencies associated with each operation of the interface may be represented as selection predicates[9]. In the engineering modelling, the transparency requirements expressed in the computational specification are mapped onto the transparency support objects. This kind of transformation, called *elimination of selection predicates*[3], is supported in LOTOS formalism. Interactions of the computational specification for which constraints on the values exchanged are explicitly imposed are replaced by unconstrained interactions in the engineering specification.

8.0 VIEWPOINT TRANSFORMATION: WHY

This section presents the benefits of viewpoint transformation approach to ODP systems design.

The computational model is an abstraction of a distributed application in terms of its requirements and expectations from the supporting engineering model.

The engineering model has to show how transparency (and other support facilities) can be achieved. A combination of two techniques can be defined [25]:

1. linking transparency mechanisms into the access path to an interface to intercept effects due to distribution (such as replication, concurrency) and take appropriate action to replace the requirements by mechanisms before passing interactions onto the object they protect.
2. adding extra functionality needed to achieve transparency by including (binding) appropriate modules at compilation time.

The engineering model of ODP is concerned not just with run-time structures and protocols, but also with the tools used to assemble, compile and link programs. The two techniques described rely on specifying application programs in an abstract form (or at least those parts of them affected by distribution) and using automated tools to transform this abstract form into a concrete form composed of a configuration of all the supporting infrastructure objects. Such an approach has significant benefits:

1. applications are easier to write because distribution is declarative: source is labelled with attributes requesting that particular transparencies and constraints be applied to selected interfaces rather than mixing application code with calls to low-level system procedures; the engineering is separated from the application.
2. applications are less error-prone because distribution details are automated.
3. programmers are more productive because they concentrate on applications rather than distribution details.
4. specifications are future proof because the rigorous and simple semantics of ADTs will survive automated changes in their representation.

The concept of viewpoint transformation permits one to look beyond. One could imagine to have compilers automatically generating:

1. stubs for access and location transparencies.
2. concurrency control managers and associated objects and mechanisms for the support of transaction or concurrency transparency.
3. group managers for the realization of replication transparency.
4. scheduling and quality of service (QoS) negotiation for real-time constraints.
5. various fault-tolerant mechanisms (including groups) based on dependability constraints.

9.0 FUTURE DIRECTION:

9. Transparencies associated with computational operations may be represented in a number of ways. This needs further exploration.

Our work extends in two directions. The first issue is finding the right set of QOS/environment attributes that are needed for some specific distributed applications that would run on ODP platform and on extending the computational model described in LOTOS with appropriate environment constraint declarations. Such an integration in LOTOS would provide an application designer/specifier with a clear division between control aspects, that are handled by LOTOS specification, and infrastructure guarantees (such as QOS/environment attributes) that appear *declaratively*. Some QOS declarations are mere assertions while some others are constraints that must be fulfilled by the infrastructure. Typically, such constraints, once formally expressed, may lead, through transformation, to possibly semi-automatic generation of QOS monitors, transparency support objects etc. We intend to explore the infrastructural requirements for the IN services/features that could be considered as applications on future intelligent networks. Specification of the computational interaction between IN services/features corresponds to the computational specification of IN services. Transformations may then be applied on the computational specification to obtain the engineering support objects which animate the (desired) computational interactions.

Having defined the computational model for distributed applications, the second direction of research is finding suitable techniques for mapping (or transforming) the computational model into engineering infrastructure. Specifically the issue of how to integrate environment constraints in the computational interface template such as to simplify the transformation exercise to the engineering realization. The existence of a formal semantics for LOTOS provides us with rigorous correctness preserving engineering solution.

10.0 CONCLUSION:

Specification transformation finds many applications. The more common is the synthesis of a protocol from service specification. Transformations are used to ensure correctness of refinement steps and can be done manually or semi-automatically. Each transformation introduces new information or takes refined decisions. Transformations may also serve as a description of the design process.

LOTOS is based on models for which (some) theory already exists for supporting transformations, e.g., in principle, a protocol can be developed from service using transformations.

The viewpoint transformation approach permits the identification of platform modules in the engineering viewpoint that support the interaction at the interfaces of Computational objects. The identification of platform modules facilitates software portability and reusability and therefore it is very strategic in many domains such as TINA, TMN, and IN which can be considered as large ODP systems [26], [27], or as applications on top of ODP platform [28].

Another important application of the viewpoint transformation approach is the very idea of the realization of the AIN “service engineering” concept: designing services at very high level starting from reusable components could be generalized in the future to a service engineering methodology supporting construction of new services by instantiation, composition and integration of existing ones. The resulting engineered service provides support for the application interaction requirements specified as part of the environment constraints (transparencies, QoS) in the computational interface templates.

A distributed application constructed using ODP computational model is executed by first *transforming* it into an equivalent engineering description. It is desired that this transformation is performed, as much as possible, automatically by tools which specialize the general computational description into one tailored to particular configuration of supporting objects.

ACKNOWLEDGEMENT: We would like to express our sincere thanks to Mr.Jean-Bernard Stefani, CNET France, Dr.Andrew Herbert, APM Ltd., Cambridge, U.K., and to Dr. Kerry Raymond, University of Queensland, Australia for the numerous discussions we have had with them on the issues related to the topic.

REFERENCES:

[1] ISO/IEC JTC1/SC21/WG7 10746-1: Basic Reference Model for Open Distributed Processing-Part 1: Committee Draft 1992.
[2] Schot, J. The Role of Architectural Semantics in the Formal Approach of Distributed System Design, Ph.D Thesis, University of Twente, Enschede, Netherlands, 1990.
[3] Bolognesi,T. (Eds): Catalogue of LOTOS Correctness Preserving Transformations, Ref: Lo/WP1/T1.2/N0045/V03, LOTOSPHERE Consortium, April 1992.
[4] ISO 8807: LOTOS: Language of Temporal Ordering Specification: A Formal Description Technique.
[5] ISO/IEC JTC1/SC21/WG7 10746-3: Basic Reference Model for Open Distributed Processing-Part 3: Committee Draft 1992.
[6] Stefani,J.,B. Towards a Reflexive Architecture for Intelligent Networks, Proceedings of the Second International Workshop on Telecommunication Information Networking Architecture (TINA), Chantilly, France, 1991.
[7] Schill,A. Zitterbart,M. A Systems Framework for Open Distributed Processing, Proceedings of the International Workshop on Distributed Systems: Operations and Management, 1992.
[8] Stefani,J.,B. Open Distributed Processing: The Next Target for the Application of Formal Description Techniques, Proceedings of the IFIP Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'91, Sydney, Australia, November 1991.
[9] Bregant,G. Platform Modelling Requirements from the ROSA Project, Proceedings of the Third International Workshop on Telecommunication Information Networking Architecture (TINA), Narita, Japan, 1992.
[10] Proctor,S. An ODP Analysis of OSI Systems Management, Proceedings of the Third International Workshop on Telecommunication Information Networking Architecture (TINA), Narita, Japan, 1992.
[11] Watson,A. ISA Project Report: Types and Projections, Ref: APM/RC.258.03, Architecture Project Management Ltd., Cambridge, U.K., April 1992.
[12] Linington,P.,F. Introduction to the Open Distributed Processing Basic Reference Model, Proceedings of the IFIP International Workshop on Open Distributed Processing, Berlin, Germany, October 1991.
[13] ISO 7498: Basic Reference Model for Open System Interconnection
[14] Langerak,R. Decomposition of Functionality: A Correctness Preserving LOTOS

Transformation, Proceedings of the IFIP Tenth International Symposium on Protocol Specification, Testing, and Verification, Ottawa, Canada, June 1990.

[15] Vissers,C.,A. FDTs for Open Distributed Systems, A Retrospective and a Perspective, Proceedings of the IFIP Tenth International Symposium on Protocol Specification, Testing, and Verification, Ottawa, Canada, June 1990.

[16] Ichikawa,H. Yamanaka,K. Kato,J. Incremental Specification in LOTOS, Proceedings of the IFIP Tenth International Symposium on Protocol Specification, Testing, and Verification, Ottawa, Canada, June 1990.

[17] van Eijk,P. Schot,J. An Exercise in Protocol Synthesis, Proceedings of the IFIP Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'91, Sydney, Australia, November 1991.

[18] Probert,R.,L. Saleh,K. Synthesis of Communication Protocols: Survey and Assessment, IEEE Transactions on Computers, Vol.40, 1991.

[19] Taylor,C.J. Object-Oriented Concepts in Distributed Systems, Computer Standards and Interfaces, 1993.

[20] Stefani,J.,B. On the Notion of Trader in Intelligent Networks, Proceedings of the First International Workshop on Telecommunication Information Networking Architecture (TINA), Lake Mohonk, USA, 1990.

[21] Rees,O. ANSA Technical Report: Using Path Expressions as Concurrency Guards, Ref: TR.022.00, Architecture Projects Management Ltd., Cambridge, U.K., February 1993.

[22] Vissers,A. Scollo,G., Alderden,R.,B., Schot,J., Pires,L.F., The Architecture of Interaction Systems: The Structuring of Distributed Systems, Lecture Notes, Enschede, Netherlands, February 1989.

[23] Pavon,S. Hultstrom,M. Quemada,J. Frutos,D. Ortega,Y. Inverse Expansion, Proceedings of the IFIP Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'91, Sydney, Australia, November 1991.

[24] Rudkin,S. Inheritance in LOTOS, Proceedings of the IFIP Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'91, Sydney, Australia, November 1991.

[25] Herbert,A. The Challenge of ODP, Proceedings of the IFIP International Workshop on Open Distributed Processing, Berlin, Germany, October 1991.

[26] Bregant,G. Towards a Convergence between Telecommunication Services Architecture and Open Distributed Processing, Proceedings of the IFIP International Workshop on Open Distributed Processing, Berlin, Germany, October 1991.

[27] Boyd,T. Telecommunication Networks as a Distributed Application Architecture: An Overview, Proceedings of the First International Workshop on Telecommunication Information Networking Architecture (TINA), Lake Mohonk, USA, 1990.

[28] Herbert,A. Green,H. Intelligent Networking as an Application of Open Distributed Processing, Proceedings of the First International Workshop on Telecommunication Information Networking Architecture (TINA), Lake Mohonk, USA, 1990.

[29] Sinderen,M.V. Schot,J. An Engineering Approach to ODP Systems Design, Proceedings of the IFIP International Workshop on Open Distributed Processing, Berlin, Germany, October 1991.