INTRODUCTION TO ODP COMPUTATIONAL MODEL

BY

KAZI FAROOQUI,

LUIGI LOGRIPPO,

DEPARTMENT OF COMPUTER SCIENCE,

UNIVERSITY OF OTTAWA,

OTTAWA K1N 6N5, CANADA.

E-mail: farooqui@csi.uottawa.ca, luigi@csi.uottawa.ca

This paper presents a tutorial introduction to the major features of the RM-ODP computational model [1-4]. The elements of the computational model are introduced and briefly explained. The model is described as an object-oriented framework of distributed applications. Major aspects of the computational model as an *interaction model*, a *construction model*, and as a distributed *programming model* are presented.

1. What is a computational model

The ODP computational model is a framework for describing the structure, specification and execution of the (components of the) distributed application on the distributed computing platform.

The computational model provides a set of basic (abstract) concepts and elements for the construction of a distributed programming (specification) language for which the model does not provide any syntax. Using the computational model, one can specify (program) a distributed application without worrying about the details of the underlying distributed execution platform (the engineering model). The design principle of the computational model is to minimize the amount of engineering detail that the application programer is required to know, yet at the same time allowing the programmer to exploit the benefits of distributed computing.

The computational model focuses on the organization of applications into distributable components, identification of interactions between application components, and the identification of the distribution requirements (from the underlying distributed execution environment) for the support of interactions between application components.

The *computational specification* of a distributed application consists of the composition of *computational objects* (which represent application components) interacting, by operation invocations, at their interfaces. It identifies the activities that occur within the computational objects, and the interactions that occur at their interfaces.

2. Computational model: An Object-Oriented view of distributed application

The computational model is based on a *distributed-object model*. It prescribes an object-oriented view of the distributed application. Applications are collections of interacting objects. In this model, objects are the units of distribution, encapsulation, and failure.

The computational model is an ‘object world’ populated with concurrent (computational) objects interacting with each other, in a *distribution-transparent* abstraction, by invoking operations at their interfaces [5]. An object can have multiple interfaces and these interfaces define the interactions that are possible with the object.

Activity is a unit of concurrency within an object. A collection of (computational) objects may have any number of activities threading through them. The *state* encapsulated by the object can be accessed and modified by the activities executing the operations in the interfaces of that object [6].

A distributed computation progresses by operation invocations at object interfaces. The *activity* in an object (invoking object) can pass into another object (invoked object) by invoking *operations* in the interface of the invoked object. Activities carry the state of their computations with them, i.e., when an activity passes into an operation it carries the parameters for that invocation, and returns carrying results. In the computational model, concurrency within an object and communication between objects are separate concerns. While concurrency is modelled by the concept of activity, communication between object is modelled as (remote) invocation of an operation [6].

The computational model provides a view of the underlying ODP platform as a distributed, multi-tasking abstract machine supporting (concurrent) objects and interactions between objects.

3. Distribution Issues in the Computational Model

The computational model places few constraints on the extent to which application programs can be distributed. Most of the constraints on distribution of application components stem from discussion in other projections, such as enterprise viewpoint or information viewpoint.

Computational specifications are intended to be distribution-transparent, i.e., written without regard to the specifics of a physically distributed, heterogeneous environment. However, the expression of *environment constraints* in the computational interface template provides a hint of the application requirements from the distributed platform, e.g., distribution transparencies, security mechanisms, specific resource requirements, etc.

At the computational level, user applications are unaware of how the underlying distributed platform is structured or how the distribution enabling and regulating mechanisms are realised.

4. Elements of the Computational Model

The design philosophy of the computational model has been to find the smallest number of concepts (elements) needed to describe distributed computations and to propose a *declarative* approach to the formulation of each concept [7].

The basic elements of the computational model are: *computational object*, *computational interface*, *operation* invocation at computational interface, *activities* that occur within a computational object, *environment constraints* on operation invocation, etc.

This section is a brief introduction of these basic computational elements out of which the *computational specification* of the distributed application is constructed. The definitions are introduced in terms of the *templates* (specification) of the corresponding elements.

4.1 Activity: Activity is agency by which computations make progress [6]. It is the unit of concurrency of the computational object. A computational object may have multiple activities threading through it, of which one or more may actually be executing on a processor at any one instant, depending upon the number of processors available. An activity may pass from one object to another by the first invoking an *operation* on the interface of the second. Activities may split into parallel sub-activities and later recombine. New activities can be initiated to proceed in parallel, independent of their initiating activity.

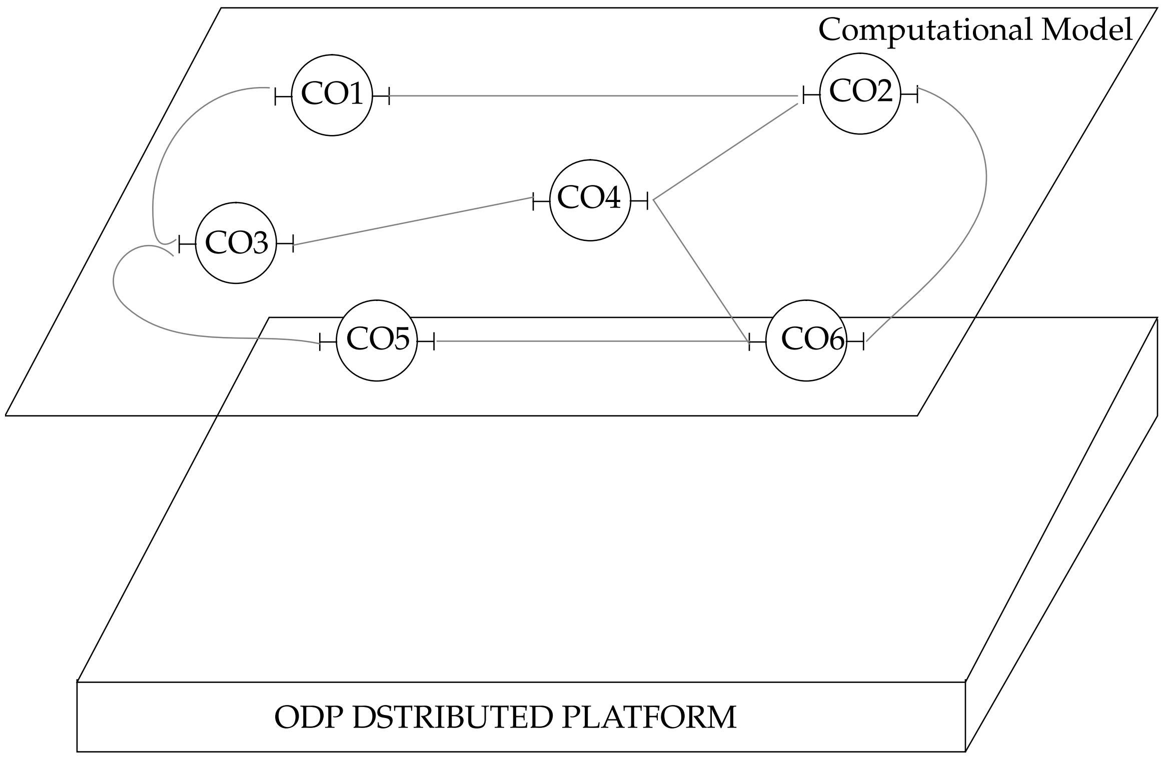

Figure 1. ODP COMPUTATIONAL MODEL: An object world supported by distributed platform (engineering model).

4.2 Computational Operation: Computational objects may support multiple interfaces as *service* provision points. A *service* is an association between object state (some data) and the programs that operate upon them [6]. The ways that a user can interact with a service are completely defined by the set of *operations* that the service supports. Operations affect the state of the object. An operation is a service primitive. Each operation has two parts: the *operation signature* which defines how the operation is invoked by a user of the service (*client*), and the *operation body*, which is the piece of program code executed by the provider of the service (*server*) when that operation is invoked.

An *operation signature* template has three parts [6]:

1. The *operation name* is an intrinsic part of the operation. When a client wishes to invoke an operation in a particular server interface it identifies it by its name within that interface. To ensure that there is no ambiguity, no two operations in the same service may have the same name.

2. The *parameter part* of an operation specifies the number and types of the parameters and the order in which they are passed to the operation when it is invoked.

3. The *result part* of an operation specifies the number and types of result for each possible outcome from the operation.

Operations have distinct outcomes, each of which can convey different numbers and types of results. An operation's possible outcomes are called *terminations*, and are distinguished by their names. For convenience one outcome from each operation can be left unnamed; this is called the *anonymous termination*, and is conventionally used to represent the normal or expected outcome, while named terminations are often used to represent unusual or unexpected results.

In the computational model, the (engineering) infrastructure failures in invoking an operation on a (remote) interface are reported (to the clients) by the infrastructure objects through the use of termination mechanism. This permits the detection of invocation failures in the infrastructure.

4.3 Computational Interface: While computational objects are the units of structure and encapsulation of (application-specific) services, interfaces are the units of provision of services; they are the places at which objects can interact and obtain services.

The distributed application components (modelled as computational objects) may be written in different programming languages and may run on heterogeneous environments. In order for a component to be constructed independently of another component with which it is to interact, a precise specification of the interactions between them is necessary. The specification of interaction between application components and of their requirements of distribution are captured in computational interfaces.

The computational interfaces model different interaction concerns of a computational object. An application component acting as a client may request a number of other components to perform operations and thus needs a different interface with each of these. Similarly, the application component acting as a server may perform actions requested by a number of client components. There is no reason to restrict a server to provide interfaces with identical specifications to each of its clients. Allowing a server to provide multiple interfaces with distinct specifications enables a computational specification to directly reflect the different roles identified in the enterprise specification, especially with regard to access control. Multiple interfaces also enable knowledge of other components to be more tightly scoped [7].

A computational object may support multiple computational interfaces which need not be of the same type. Interfaces of the same type may be provided by objects which are not of the same type. Each object may provide interfaces which are unlike those provided by the other object.

In the ODP computational model, interactions are specified in terms of either operational or non-operational interfaces.

1. Operational Interface

2. Non-operational Interface.

4.3.1 Operational Interface: The specification of an operational interface template consists of [3]:

1. Operation Specification

2. Property Specification

3. Behavior Specification

4. Role Indication

Operation Specification: The definition of operations that are supported by the interface. Operation specification includes:

1. Operation name: Each operation has a local name within an interface template. No two operations, within the interface, may have the same name.

Data Specification:

2. The number, sequence, and type of arguments that may be passed in each operation.

3. The number, sequence, and type of results that may be returned in each termination.

The operation name together with the type of argument and result parameters constitutes the *operation signature*. Both the operation names and the arguments can be represented as abstract data types.

Most interface specifications, to date, have concentrated on the syntactic requirements of the interface such as the operation signature. Aspects other than pure syntax are also important in facilitating the interaction between a pair of objects. This additional semantic information falls into two categories [8]:

* information affecting the way in which the infrastructure supports the interactions; this information constrains the type of distribution transparencies, choice of communi-

cation protocols, etc. that must be placed in the interaction path between the interacting objects.

* the behavior (or the semantics) of the service offered at the interface; an interface is viewed as a projection of an object's behavior, seen only in terms of a specified set of observable actions. As a result, signature compatibility is less discriminating than interface compatibility.

Property Specification: The property specification in the computational interface template defines the following attributes:

1. distribution transparency requirement on operation invocation.

2. quality of service (including communication quality of service) attributes associated with the operations.

3. temporal constraints on operations (e.g., deadlines).

4. dependability constraints (e.g., availability, reliability, fault tolerance, security etc.)

5. location constraints on interfaces (and hence their supporting objects).

6. other environment constraints on operations (e.g., those arising from enterprise and information viewpoint).

These attributes may be associated with individual operations or the entire interface. Property specification is an important component of the computational interface template and has a direct relationship to the realized engineering structures and mechanisms.

Behavior Specification: It defines the behavior exhibited at the interface. All possible orderings of operation invocations at or from this interface can be specified. This includes ordering and concurrency constraints between operations as well as sequential and parallel operation invocations. The behavior constitutes the protocol part of the interface.

Role Indication: In general, an interface specification may be bi-directional and specify the operations each of a pair of application components could request the other to perform. For simplicity, the ODP computational model only contains uni-directional interface specifications which directly support client-server interaction.

Often an object assumes the role of either *client* or *server*. All interactions of an object, both as a client and as a server, between it and its environment occur at object interfaces. It is convenient to partition client-role interaction concerns from server-role interaction concerns in different interfaces.

4.3.2 Non-operational Interface: The computational objects may perform both the information processing task as well as act as containers of information. There is a need to model not only the interfaces which provide ‘service’, but also those interfaces which model ‘continuous’ information flow. Such interfaces are modelled, in the computational model, as *non-operational interfaces* (also known as *stream interfaces)*.

The non-operational interface is a set of information flows whose behavior is described by a single action which continues throughout the life time of the interface. Information media such as voice and video inherently consists of a continuous sequence of symbols. Such media are described as *continuous* and the term *stream* is used to refer to the sequence of symbols comprising such a medium [9].

Examples include the flow of audio or video information in a multimedia application, or the continuous flow of periodic sensor readings in a process control application. The computational description does not need to be concerned with detailed mechanisms; the fact that the flow is established and continues during the relevant period is enough.

The template for a non-operational or stream interface consists of:

Stream Signature: A specification of the type of each information flow contained in a stream interface and, for each flow, the direction in which the flow takes place.

Environment Constraint: Continuous media have strict timing and synchronization requirements. The environment constraints that are relevant to stream interfaces include synchronization and clocking properties, time constraints, priority constraints, throughput, jitter, delay, media-specific communication quality requirements, etc., in addition to the properties applicable to operational interfaces.

Role: A role for each information flow, e.g., a producer object or a consumer object.

4.4 Computational Object: The components of a distributed application are represented as computational objects in the computational model. The computational objects are the units of (application) structure and distribution. A computational object is an encapsulation of (an application-specific) state and mechanism which are not directly accessible to any other object. The computational objects model both the application components that perform information processing and those components that store the information. Objects can create interfaces or stop them during their lifetime.

A computational object template consists of:

1. a set of computational interface templates (both operational and stream) which the object can instantiate.

2. an action template for initializing the state of new instances of the object.

3. a specification of environment constraints applicable to the object as a whole.

A computational object can perform the following *activities* [3]:

1. instantiation of interface templates (creating an interface),

2. instantiation of object templates (creating an object),

3. trading for an interface,

4. binding to an interface,

5. invoking an operation at an operational interface,

6. reading and writing the state of the object,

7. spawning, forking, and joining actions,

8. stopping of interfaces,

9. stopping of object.

These basic actions can be composed in sequence or in parallel.

5. Multiple views on Computational Model

There are several ways in which the general computational model can be described. This section identifies the major aspects of the computational model. The computational model can be viewed as:

1. *interaction model* - an environment for interaction between computational objects.

2. *construction model* - construction of configuration of computational objects.

3. *programming model* - an application programming environment.

Together, these aspects address the issues related to the functional decomposition of the distributed application, inter-working, and portability of application components.

5.1 Interaction Model: One view of the computational model is as an environment that supports the existence of and the interaction between computational objects. Computational objects interact by invoking operations at their interfaces. The interaction model defines an *invocation scheme* and a *type scheme* [10].

The invocation scheme describes the permitted forms of interaction, i.e., how clients may use the interfaces provided by the server. It defines the mechanisms for parameter passing between interfaces.

The type scheme provides a set of types into which computational interfaces can be classified. It defines a conformance relation over interface types which are a set of matching rules between interfaces which must be satisfied before a binding between interfaces can be established.

The interaction model (invocation scheme) is simple and uniform. It is based on the concept of *operation* invocation. The interaction between computational interfaces is via operation invocations which carry input argument parameters and the result of operation execution is returned to the invoker of the operation via *termination*.

The interaction model (invocation scheme) supports two styles of interactions between computational objects (or more precisely between computational interfaces): *interrogations* and *announcement*: to model interactions with and without the reply respectively.

Interrogation is a synchronous *request-response* invocation style; the *activity* that invoked the interrogation passes (via *operation*) to the object that provides the invoked operation, and subsequently returns (via termination) to the object from which the invocation was made. There is no change in the degree of concurrency of the system using an interrogation type of invocation.

Announcement is an asynchronous *request-only* invocation style; a new *activity* is created in the object that provides the invoked operation, and the invoking activity continues in the object from which it made the invocation. Invoking an announcement increases the concurrency in the system, the completion of the evaluation of the body of an announcement decreases the concurrency in the system. The object that invoked the announcement is informed neither of the completion of evaluation (of body of operation) nor of the results delivered (if any).

The interaction model is independent of the kind of computational objects that participate in the interaction as well as of the way in which a computational object has been structured internally. The interaction model thus supports the notion of encapsulation and information hiding [11]. This model establishes the interpretation of parametrization and gives failure semantics for the interaction [12].

5.2 Construction Model: The construction model is concerned with the construction of the configuration of computational objects, and supports the creation of complex networks of interacting objects, giving the rule which govern object composition and decomposition [12].

The computational objects can be connected in various ways, and networks of such objects can be treated as a single computational object. Similarly, a single computational object can be decomposed into networks of computational objects [11].

5.3 Programming Model: The computational model provides an abstract, distribution-transparent, language-independent *specification* and *programming model* for distributed applications, and of their execution and interaction semantics. Concerns in this viewpoint essentially include specification/programming language and programming system interface issues. The computational model expresses the *programmability* of the distributed platform [13].

The language-independent *programming framework* offered by the computational model provides:

1. Application programming interfaces (APIs).

2. Programming concepts and abstractions necessary for the development of distributed applications (an abstract programming language).

From this viewpoint an ODP system appears as a large programming environment capable of building and executing applications on the supporting engineering infrastructure. The distributed programming model provided by the computational model, abstracts away, in an integrated framework, the generic set of distributed services provided by the engineering model from distributed applications designers and programmers. The ODP engineering model, that describes the structure and organization of these distribution enabling and regulating services, constitutes a *virtual machine* model for executing distributed programs conforming to the ODP computational model [14].

Hence, the computational model provides the equivalent of programming language, for use on top of the *abstract machine* realized by the engineering infrastructure. Such a computational model will contain programming language features which are commonly found in advanced object-based distributed platforms. As such, the computational model can be seen as some form of implementation language for building applications on top of ODP systems [15].

The ODP computational model is based on a remote procedure call and a lightweight threads style of programming. This can very easily be noted since, in the computational model, all object interactions are considered remote and the invocation of interrogation corresponds to the procedure call. Such a style is a natural extension of the procedural style found in the majority of programming languages.

The computational model hides the actual degree of distribution of an application from its programmer, thereby ensuring that application programs contain no deep-seated assumptions about which of their components are co-located and which are separated. Because of this, the configuration and degree of distribution of the underlying platform on which ODP applications are run can easily be altered without having a major impact on the applications software [16]. This desirable characteristic is called *distribution transparency.*

Since the main objective of ODP is to provide a generic architecture for distributed systems, the role of the computational model is particularly important. The computational model masks the effects of distribution and heterogeneity, when required from applications.

By conforming to the computational model, application programmers are given a guarantee that their programs will operate in a variety of different quality environments, without modification of the source. The engineering model offers standardized system programming interfaces for supporting the computational programming environment [17].

6. Conclusion

The computational model concentrates on the problems and the opportunities presented by the execution of application components on distributed computing systems. It identifies the functions that must be available to the programmer and the constrains on the application structure necessary to enable distribution, rather than a particular syntax of the computational language. The outcome of this approach is that all programs, in whatever language, are written with the same abstract (distributed) machine as their target. As suggested in [11], porting a program from one system to another is then a matter of only changing the local representation of the abstract machine as it appears in the application programming language, which does not require any changes to the application program itself.

7. References

[1]. Draft Recommendation ITU-T X.901 / ISO 10746-1: Basic Reference Model of Open Distributed Processing - Part-1: Overview.

[2]. Draft International Standard ITU-T X.902 / ISO 10746-2: Basic Reference Model of Open Distributed Processing - Part-2: Descriptive Model.

[3]. Draft International Standard ITU-T X.903 / ISO 10746-3: Basic Reference Model of Open Distributed Processing - Part-3: Prescriptive Model.

[4]. Draft Recommendation ITU-T X.904 / ISO 10746-4: Basic Reference Model of Open Distributed Processing - Part-4: Architectural Semantics.

[5]. S.Proctor, “An ODP Analysis of OSI Systems Management”, Proceedings of the Third Telecommunication Information Networking Architecture Workshop, (TINA 92), Narita, Japan, January 1992.

[6]. ANSA: An Application Programmer’s Introduction to the Architecture, TR.017.00, Advanced Projects Management Limited, Cambridge, U.K., November 1991.

[7]. ANSA: An Engineer’s Introduction to the Architecture, TR.03.02, Advanced Projects Management Limited, Cambridge, U.K., November, 1989.

[8]. ANSA Reference Manual, Volume C., Release 01.01, Advanced Projects Management Limited, Cambridge, U.K., July 1989.

[9]. ANSA Technical Report: Integrating Multimedia into ANSA Architecture, TR.028.00, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[10]. ANSA Computational Model, AR.001.01, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[11]. ANSA Reference Manual, Volume A., Release 01.01, Advanced Projects Management Limited, Cambridge, U.K., JUly 1989.

[12]. P.F.Linington, "Introduction to Open Distributed Processing Basic Reference Model", Proceedings of the IFIP TC6/WG6.4 International Workshop on Open Distributed Processing (October 1991), North Holland 1992.

[13]. G.Bregant, "Platform Modelling Requirements from the ROSA Project", Proceedings of the Third Telecommunication Information Networking Architecture Workshop, (TINA 92), Narita, Japan, January 1992.

[14]. J.B.Stefani, E.Najm, "A Formal Semantics for the ODP Computational Model", to appear in the CN&ISDN special issue on Open Distributed Processing.

[15]. J.B.Stefani, "Open Distributed Processing: The Next Target for the Application of Formal Description Techniques, Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE'90.

[16]. ANSA Technical Report: DPL Programmers Manual, TR.031.00, Advanced Projects Management Limited, Cambridge, U.K., February 1993.

[17]. J.B.Stefani, "Towards a Reflexive Architecture for Intelligent Networks", Proceedings of the Second Telecommunication Information Networking Architecture Workshop, (TINA 91), Chantilly, France, March 1991.