AN INFORMATIONAL OBJECT MODEL FOR ODP APPLICATIONS

Bouabid El Ouahidi and Mohamed Bouhdadi
Mohamed V University, Faculty of Science
Department of Mathematics and Computer Sciences
P. O. 1014 Rabat Morocco
Tel. and Fax: 00 (212) 7 77 54 71
email: {ouahidi, bouhdadi}@fsr.ac.ma

ABSTRACT

The Reference Model-Open Distributed Processing (RM-ODP) provides a framework for the standardization of Open Distributed Processing (ODP). It defines an object model and an architecture for the construction of ODP systems in terms of five viewpoints. However, the RM-ODP is abstract and therefore cannot be easily applicable. Indeed, several issues must be addressed. The objective of this paper is twofold. Firstly, based on RM-ODP itself, we define a concrete typing system for the ODP information objects. This model is a contribution for defining the ODP type repository function. Secondly, we show that the Object Constraint Language (OCL) can be used for the ODP type descriptions, and for the ODP information viewpoint specifications. Using OCL we apply that typing system to describe the ODP trading information viewpoint.

Keywords: ***RM-ODP, Object Model, Information, Description Language, OCL, Trading***

1.0 INTRODUCTION

Distributed processing is rapidly expanding as it allows increasing the performance, the evolution, and the use of existing systems. However, it is very difficult to benefit from distributed processing without any support because of its inherent properties such as concurrency, asynchronism, transactions, and so on. Furthermore, the heterogeneity of computer systems makes this task even more difficult. This heterogeneity includes the heterogeneity of programming languages, operating systems, hardware, communication protocols and also the heterogeneity of the application domains. The object-oriented approach would permit masking these heterogeneities. The OMG (Object Management Group) consortium is working towards defining an architecture, OMA [1] whose objective is to define the object-oriented concepts to ensure the interoperability of applications running on heterogeneous systems. It defines the architecture CORBA (Common Object Request Broker Architecture) [2] whose core is the ORB (Object Request Broker) that realizes the communications between client objects and server objects by brokering requests between them. It also defines a universal language for the definition of interfaces (IDL) (Interface Definition Language) [2]. IDL masks the heterogeneity of programming languages by developing compiler IDLs to other programming languages [2]. CORBA is in fact an integration of the client-server paradigm and the object-oriented paradigm. It is the specification, which allows the invocation of an operation on a distant object independently of the localization and programming language used. CORBA specifies the implementation of the object oriented distributed applications but does not specify how to design these applications. However, the objective of the inter-working of heterogeneous applications is not effective unless it is considered in the overall process of development. This is the aim of the Open Distributed Processing. The RM-ODP [3, 4, 5, 6] provides a framework within which supports of distribution, interworking and portability can be integrated. It defines an object model and an architecture for the construction of open distributed systems. The object model defines concepts for information and processing. The architecture defines five viewpoints, which are enterprise viewpoint, information viewpoint, computational viewpoint, engineering viewpoint and technology viewpoint. Each viewpoint handles a particular aspect of an ODP system. The architecture also defines a viewpoint language for each viewpoint, the ODP functions and the ODP transparencies.

Elsewhere, in order to be applied in a specific domain, RM-ODP must be extended and specialised. For instance, the TINA (Telecommunication Information Network Architecture) [7, 8] is based on RM-ODP.

However, the RM-ODP is abstract and does not constitute a methodology itself, and hence cannot be directly applicable. It only provides a framework for the definition of ODP standards. These standards include standards for ODP functions; standards for modelling and specifying ODP systems; standards for methodology, programming, implementing, and testing of ODP systems. Furthermore, the RM-ODP recommends to define concrete types of information to use in the viewpoint specifications. The type repository function constitutes an important subject of standardization for the ISO/ITU-T WG7 Group.

Several issues must be addressed to construct ODP systems [9, 10]. Current researches are focussing on different aspects such as the applicability of UML (Unified Modelling Language) [11, 12] to develop ODP systems, and in particular the ODP enterprise specification [13, 14, 15, 16, 17].

The UML language is rapidly emerging as the de-facto standard for modelling Object-Oriented (OO) systems. Given this role, it is imperative that the UML needs a well-defined, fully explored semantics. Such semantics is required in order to ensure that UML concepts are precisely stated.

Whereas grammars are well suited for text, the UML meta-model works well as a description of the structure of UML grammars [18]. The UML meta-models capture a precise notion of the syntax of the UML modelling techniques (this is what meta-models are typically used for), but they do little in the way of answering questions related to the interpretation of non-trivial UML structures [19, 20, 21].

Rather than generate formal specifications from informal OO models and require that developers manipulate these formal representations, a more workable approach is to provide formal semantics for graphical modelling notations and develop rigorous analysis tools that allow developers to directly manipulate the OO models they have created. The degree of formality of a model is not necessarily related to its form of representation. This is the objective of the pUML group (Precise UML) [22].

The structure of the document is as follows. We define two UML meta-models of the ODP information concepts in Section 2. These meta-models describe the syntax of a modelling language for ODP applications. We propose a concrete information object model in Section 3. Section 4 is about the adequate formal language for ODP type descriptions; we show that the OCL (Object Constraint Language) [23] language can be used to meet this attempt. We apply this work to describe the ODP trading information viewpoint in Section 5. A conclusion and perspectives end this paper.

2.0 THE RM-ODP OBJECT MODEL

In general, the term object model refers to the collection of concepts used to describe objects in an object-oriented specification (OMG CORBA object model, RM-ODP object Model, UML meta-model object model). It corresponds closely to the use of the term data model in the relational data model. RM-ODP is a framework for the construction of open distributed systems. It defines a generic object model, a set of architectural concepts in the foundations part, and an architecture, which contains the specifications of the required characteristics that qualify, distributed processing as open. The architecture extends and specialises the concepts of the foundations part. To define concrete information object model, we will describe all the RM-ODP object concepts.

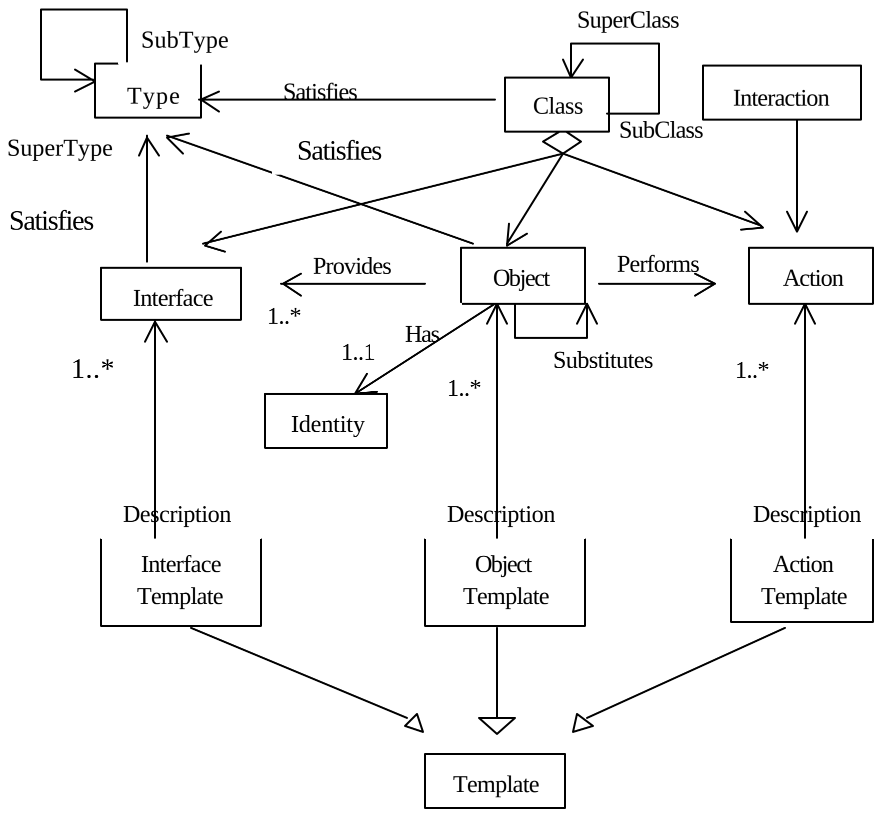


Fig. 1: RM -ODP Object Model

2.1 The RM-ODP Foundations

The object model [4] defines the basic concepts concerned with existence and activity: the expression of what exists, where it is and what it does. We describe these concepts graphically using the UML notation (Fig. 1).

The core concepts defined in the object model are object and action. An action is something, which happens. An object is a model of an entity. It is characterised by its behaviour, and dually by its states. Depending on the RM-ODP viewpoint, the emphasis may be placed on the behaviour or on the states. When the emphasis is placed on behaviour an object is said to perform functions and offer services, theses functions are specified in terms of interfaces. It interacts with its environment at its interaction points that are its interfaces. An object is distinct from any other object by its identity.

The other concepts defined in the object model are derived from the concepts of object and action; they are Class, Template, Type, Subtype, Supertype, Subclass, Superclass, Composition, and Behavioral Compatibility.

An object is behaviorally compatible with a second object if the first object can replace the second object without the environment being able to notice the difference in the object behaviour. The RM-ODP behavioral compatibility concept corresponds to the OMA substitutability concept. Composition of objects is a combination of two or more objects yielding a new object. A type (of an <X>) is a predicate characterizing a collection of <X>s. A class (of an <X>) defines the set of all <X>s satisfying a type. The type concept corresponds to the type concept of UML. The class concept corresponds to the OMG extension concept, the extension of a type is the set of values that satisfy the type at any particular time. A <X> template is the specification of the common features of a collection X in a sufficient detail that an X can be instantiated using it; the template has the meaning of C^{++} class.

Note that an object has a type, a class, and a template. It is the case for actions and interfaces.

2.2 The RM-ODP Architecture

The architecture [5] comprises: (1) five viewpoints, (2) a viewpoint language for each viewpoint, (3) specifications of functions required to support ODP systems, and (4) transparency prescriptions showing how to use the ODP functions to achieve distribution.

The distribution transparencies hide the aspects of distributed processing in a desired way from the viewpoint of the users. Each of the viewpoints handles a particular aspect of the ODP system. The enterprise viewpoint focuses on the purpose, scope and policies for the ODP system. The information viewpoint focuses on the semantics of information and the semantics of the information processing. The computational viewpoint enables distribution through functional decomposition of the system into objects, which interacts at interfaces. The engineering viewpoint focuses on the mechanisms and functions required to support distributed interaction between objects in the system. The technology viewpoint focuses on technology in that system.

The definition of a language for each viewpoint describes the concepts and rules for specifying ODP systems from the corresponding viewpoint. The object concepts defined in each viewpoint language are specializations of those defined in the foundation part of RM-ODP.

The ODP functions define the functionalities of the ODP operating system assumed to process the difficulties inherent to distribution. They are classified into categories, which include among others the repository functions related to database management functions. In the following section, we describe briefly the enterprise, information, and computational languages, the other viewpoint languages have no interest for our study.

An enterprise specification defines the purpose, scope and policies of an ODP system. A policy is a set of rules related to a particular purpose. A rule can be expressed as an obligation, a permission, or a prohibition. An ODP system consists of a set of enterprise objects. An enterprise object may be a role, an activity or a policy of the system.

An information specification defines the semantics of the information and the semantics of information processing in terms of a configuration of information objects, the behaviour of these objects and environment contracts for the system. An information object template is defined in terms of static, invariant and dynamic schema;

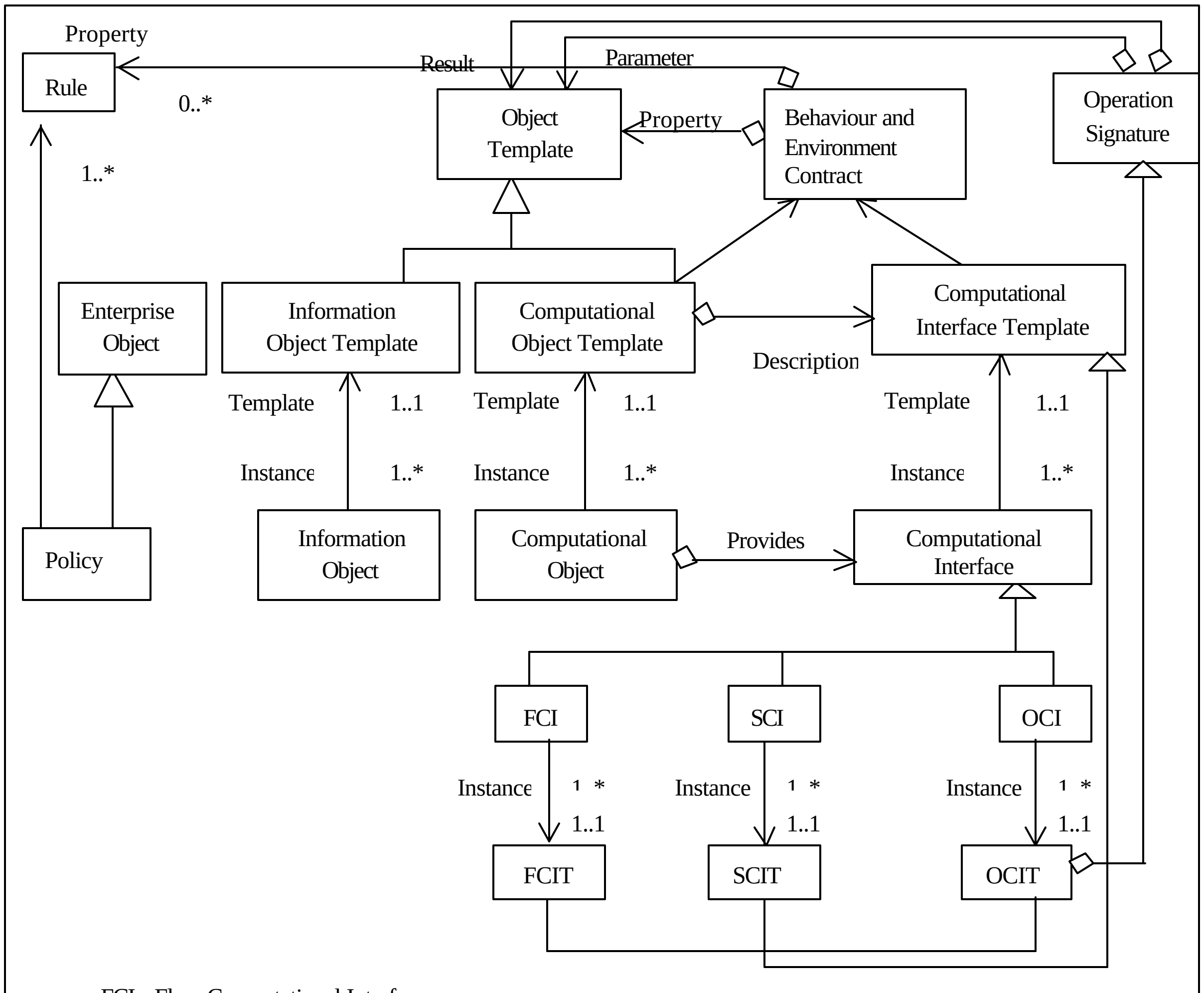
Invariant schema: A set of predicates in one or more information objects, which must always be true. The predicates constrain the possible states and state changes of the objects to which they apply.

Static schema: A set of predicates in one or more information objects, at some point in time, subject to the constraints of any invariant schema.

Dynamic schema: A specification of the allowable state changes of one or more information objects, subject to the constraints of any invariant schema.

An information object is either atomic or composite. The state of the composite object is represented by the combined state of its component information objects. The information objects resulting from the instantiation of a composite information object template only exist as part of the instantiated composite object and have no meaning outside it.

A computational specification defines the functional decomposition of an ODP system into objects, which interact at interfaces. The basic concepts of the computational language are the computational interface, the computational object, the interaction and the binding object. The binding object is a computational object, which supports a binding between a set of other computational objects. An interaction is either a signal, a flow or an operation. A signal is an atomic shared action resulting in one-way communication from an initiating object to a responding object. A flow is an abstraction of a sequence of interactions resulting in a conveyance of information from a producer object to a consumer object. An operation is an interaction between a client object and a server object. Like interaction kinds, an interface is either signal, operation or flow. A signal interface is an interface in which all the interactions are signals. In an operation interface all the interactions are operations. All the interactions are flows in a flow interface. A computational object template comprises a set of computational interface templates which the object can instantiate, a behaviour specification and an environment contract specification. The behaviour of the object and the environment contract are specified in terms of a set of properties (attributes). A computational interface template is associated to each kind of interface. It comprises a signal, a flow, or an operation interface signature as appropriate, a behaviour specification and an environment contract specification. The behaviour and the environment contract are defined as set of properties.



FCI : Flow Computational Interface
FCIT : Flow Computational Interface Template
SCI : Signal Computational Interface
SCIT : Signal Computational Interface Template
OCI : Operation Computational Interface
OCIT : Operation Computational Interface Template

Fig. 2: RM -ODP Architecture Object Model

3.0 THE ODP INFORMATIONAL OBJECT MODEL

An object is characterized by its behaviour and dually by its states. Depending on the RM-ODP viewpoint, the emphasis may be placed on the behaviour or on the states. When the emphasis is placed on behaviour, an object is said to offer services and to interact at its interfaces; this is the concept of a computational object. When the emphasis is made on states, an object does not have interfaces, it cannot interact, this is the concept of an information object.

The information object and computational object have respectively the meaning of non-object and object in the OMA core object model. Indeed, from the viewpoint of OMA, an object provides services at an interface, while a non-object is not an object, but may be used as a value. RM-ODP recommends the definition of the concrete types of information (of values) to be used in the viewpoint specifications. This is the subject of this section.

In the OMA object model, the primitive values are not objects, however in our model, these primitive values are really objects. This is the case of Java programming language, whose all-primitive types are objects.

3.1 The Classification of Information

We define in this section, the criteria of the classification of the information objects. We describe three main criteria of classification of information.

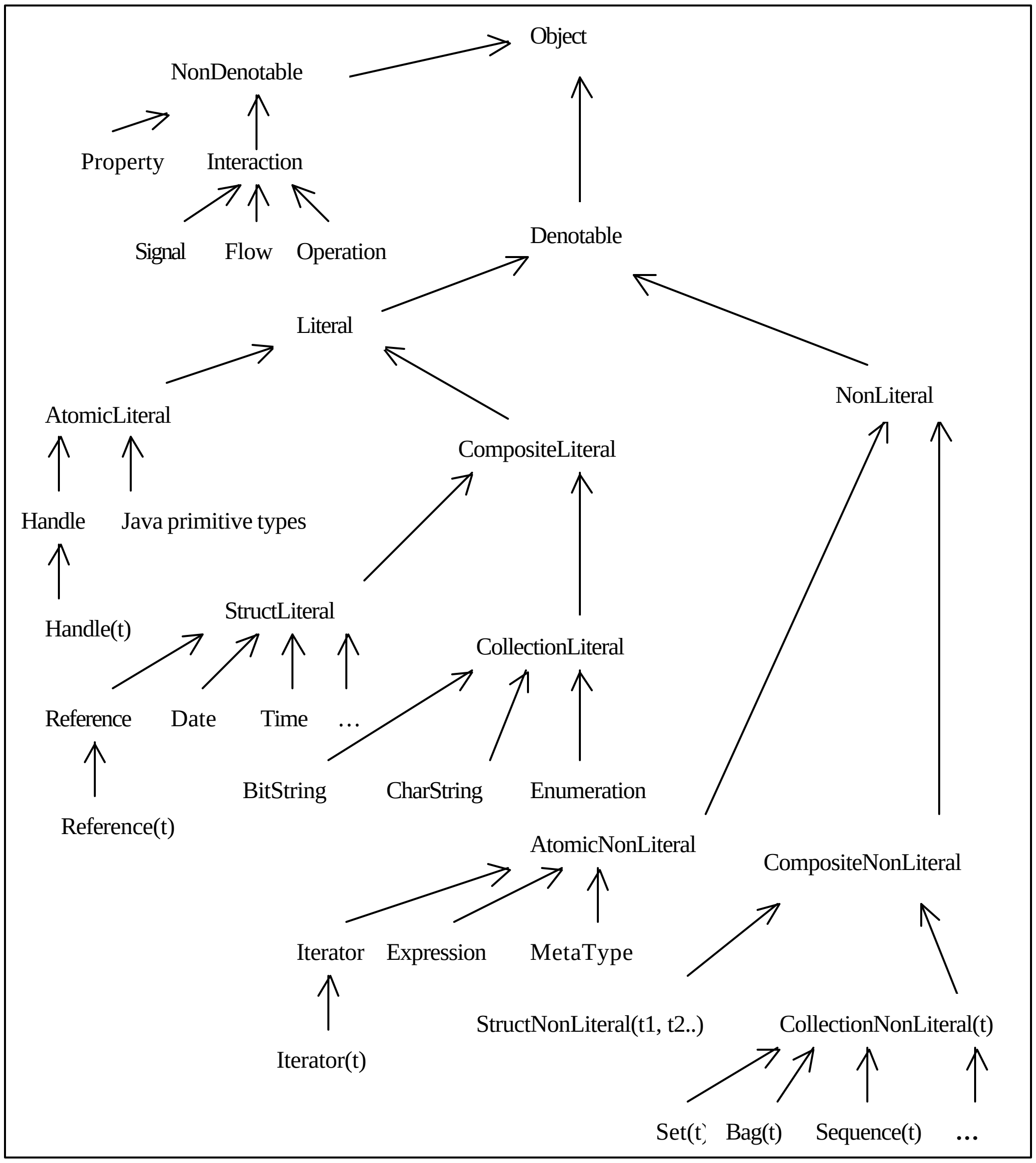


Fig. 3: The ODP Informational Object Model

Given an information object:

- Does the information object have an identity or not?
- How is the information object identity represented in the machine; that is, does the identity of the information contain the value of the information or does it only contain the reference of the value of the information? This describes the relationship between the identity and the value of the information object.
- Is the information object atomic or composite?

We will describe and justify each of these criteria, which are successively applied to build our information typing system.

The information objects can be organised into a hierarchy of sub-types and super-types; this corresponds to the concept of behavioral compatibility. A subtype inherits all the characteristics of its supertypes. An object is defined by its characteristics, which include the signature of its services, its behaviour and the contract of its environment.

An object has an intrinsic identity. A characteristic of an object is an information object that has equally an identity but this identity has no meaning outside of this object. This corresponds to the first criteria of the classification of the information objects. That is, the information objects are classified into two categories, which are the information objects that have an identity and the information objects that have no identity.

The information objects which do not have an identity (characteristics of objects) are called **NonDenotable** objects. According to the RM-ODP definition of computational interfaces, we define the NonDenotable objects. They are signal, flow, operation, and property. Note that, an information object is an aggregation of these objects. For example, The getProperties() operation of the Java Class System returns a set of properties of the system.

The information objects, which have an identity, are called **Denotable** objects. This information object must identify a local or a distant object depending on RM-ODP viewpoints. Indeed, from the engineering viewpoint, to interact with a distant object, one must have the identity of its computational interface. In RM-ODP the identity of distributed computational interface is denoted **Reference**. This type has to be included in an ODP typing system.

In the following, we do not consider the distribution aspects. The hierarchy of the Denotable objects can be defined according to the representation of the value of the identity. Indeed, the representation used to translate this identity into the machine code is not the same for all the information objects. We distinguish two categories of Denotable information objects, which are the literal (**Literal**) information objects and the non literal (**NonLiteral**) information objects.

The identity of a literal object contains exactly the value of that object. In contrast, the identity of a NonLiteral information object does not contain the value of that object, but only a value, which references the value of the object. This means that the identity of a NonLiteral object contains a handle. To this end, our system contains the type **Handle**. In fact, each object of type t has a handle of type **Handle(t)**. The type handle is an abstraction of any Handle(t). This implies that a NonLiteral object necessitates an operation, which allocates the necessary space for the representation of the value of the object and returns this space as the result of that operation. In contrast, a Literal object does not necessitate any allocation of memory since the Literal objects implicitly pre-exist.

The Java programming language illustrates; this criterion (Literal/NonLiteral) although it is a “pure” object-oriented language in the sense that everything is an object. This means, that we manipulate objects through handles and hence must create all the objects. However, for primitive types Java falls back on the approach taken by C and C^{+}: instead of creating the variable using new, an “automatic” variable is created which is not a handle. The variable holds the value itself. Most of the primitive data types have “wrapper” classes for them. That means if we want to treat a primitive type as a non-literal, we use the associated wrapper.

After applying the two previous criteria we apply the criteria of the structure of the information object such as that being described in the ODP information viewpoint. That is, is the value (of the object) atomic or composite?

We obtain:

(1) the atomic literal object (**AtomicLiteral**),
(2) the composite literal object (**CompositeLiteral**),
(3) the atomic non literal object (**AtomicNonLiteral**), and
(4) the composite non literal object (**CompositeNonLiteral**).

In the AtomicLiteral objects, we include Java primitive types, handle and handle(t).

The CompositeLiteral objects are classified into two subtypes, which are the struct literal (**StructLiteral**) objects and the collection literal (**CollectionLiteral**) objects. A struct literal object has a fixed number of named fields such that each field contains a literal. A CollectionLiteral object is a composition of literals having the same type. This constitutes the criteria corresponding to the types of the components of a composite information (that is, are they of the same type or not). The StructLiteral objects we define are **Date, Time, Time-Stamp** and **Interval** as defined in the SQL ANSI standard. The sub-types of the LiteralCollection object we define are **BitString, CharString** and **Enumeration.**

The atomic non-literal (**AtomicNonLiteral**) objects are defined according to the RM-ODP objects. Indeed, within RM-ODP, an object may be an enterprise object, an information object or a computational object from the user's viewpoint.

An enterprise object may be a use case of the system, a user of the system or a **policy** which governs the activities of the system. An information object is the data manipulated by a computational object. The policy of the system consists of a set of rules that govern the activities of the system ensuring the objective of the system. In the information viewpoint, these rules are **criteria** and **constraints**. Thus, our typing system has to include the type defining values of criteria and constraint. We will call this type **Expression**. It is a sub-type and an instance of AtomicNonLiteral.

Note that another reason to include the expression type is that the parameters of a database request are criteria and constraints and the ODP functions include the functionality of management of database systems. This necessitates the use of a SQL request as ODP-operation. This integration is very important and yet a non-traditional approach. This is the main objective of Java Database Connectivity [24].

The types are themselves objects, and hence have equally properties and operations. Each type is an instance of a type that allows manipulation of the information concerning types of objects. We call it **MetaType**.

In summary, our typing system of information includes among others the types, **Expression**, **Handle**, **MetaType** and **Reference**. However, the difficulty is how to define them.

We think that the set of instances of types **MetaType** and **Expression** can be defined only by grammars. The grammar of OCL expressions is an example [23].

An instance of MetaType or of Expression or of Handle has attributes and properties that we will make explicit. The Handle type can be defined like in C^{++} with the semantics of the creation and the copy operations.

However, making explicit the type Reference is a challenge. We think that it is a sub-type and an instance of StructLiteral.

The CompositeNonLiteral objects are classified into the struct objects (**StructNonLiteral**) and the collection objects (**CollectionNonLiteral**). The StructNonLiteral objects have a fixed number of named fields such as each field is either a literal or a non-literal object. These fields may be of different types. A CollectionNonLiteral object contains other objects that must be of the same type. Hence, we define a generic type of collection denoted as **Collection(t)**. We classify the CollectionNonLiteral objects according to two criteria; the obligation of the *order* and the permission of the *duplication* of the components of the collection objects. Applying these criteria we obtain the **Sequence**, **Array(t), Set(t)** and **Bag(t),** where Sequence and Array are ordered components; Array and Bag permit duplication. Also, we define a type of iterators of collections denoted **Iterator(t)** like in Java programming Language (the Enumeration).

3.2 Typing System

Our typing system includes types defined in many typing systems such as CORBA IDL, OCL, C, C^{++} and JAVA. The semantics of each information object can be easily defined using the OCL mechanism of pre- and post-conditions. Next, we define some of types, such as; Property, Operation, Flow, Signal, Parameter, MetaType, Object, Collection, Expression, Iterator and Handle.

When the semantic of a characteristic of a type is obvious, we define it in English language.

Property
name : CharString
type : MetaType
mandatory : Boolean
readonly : Boolean

Operation
name : CharString
typeReturn : MetaType
parameters : Collection(Parameter)

Flow
name : CharString
typeReturn : MetaType
parameters : Collection(Parameter)

Signal
name : CharString
typeReturn: MetaType
parameters : Collection(Parameter)

Parameter
name : CharString
type : MetaType
passageMode : Enumeration ("in", "out", "in/out")

The types MetaType, Object, Collection and Expression, correspond respectively to the types, OclType, OclAny, collection and OclExpression.

Iterator(t)
Next() : t
First() : t
Last() : t
HasMoreElements() : Boolean
Reset()
Delete()

The characteristics of Iterator(t) have the same meaning as the Enumeration primitive type in Java. We call iterator(t) an instance of Iterator(t) associated with an instance of Collection(t) called collection. Next() gets the next object in collection. HasMoreElements() sees if there are any more objects in collection.

Handle(t)

Create () : Handle(t)
Create(aHandle : Handle): Handle(t)
Create(aHandle : Handle(t)): Handle(t)
Copy(aHandle : Handle): Handle(t)
Copy(aHandle : Handle(t)): Handle(t)

The characteristics of Handle(t) have the same definition as the Reference type in C^{++}.

Other obvious types are described in Fig. 3.

4.0 WHY OCL IS USED FOR THE ODP TYPE DESCRIPTIONS

Several specification languages have been developed; each handles a particular aspect of a system. For instance, Z [25] and VDM [26] focus on specifying the behaviour of sequential systems; others such as CSP [27] and CCS [28] Statecharts [29] focus on specifying the behaviour of concurrent systems. SDL [30] and LOTOS [31] are not object-oriented and therefore do not support the expression of basic object concepts, transaction or multi-threading concepts. The SDL92 [32] and GDMO-GRM [33] are object-oriented, but SDL'92 focuses on specialization and inheritance while the formalism of GDMO does not cover the behavioral aspects of a system.

In fact, no formal description technique is able to describe in a complete way the ODP concepts. The inherent characteristics of ODP systems imply the need to integrate different specification languages, each specialized in a particular kind of properties and also to handle non-behavioral properties of ODP systems. It is recognized to take benefits from the well-established verification techniques; that is, to integrate the theorem proving and model checking techniques. We can therefore conclude that up to now, no formal method is likely to be suitable for specifying and verifying every aspect of an ODP system. We need to support all different kinds. Methods and tools should work in conjunction with each other. More precisely, rather than build a single method, we can build meta-method which itself produces methods customized for a particular problem domain. This represents the objective of the inter-working in the area of formal methods. Progress will depend on future directions on fundamental concepts and principles. Those concepts would include among others integration of formal methods, and integration of those with the system development process. Indeed, formal methods can complement less formal methods that are used in the overall system development process.

Elsewhere, the ISOWG7 group shows that the type repository function standard must permit the use of multiple type description languages. There are a number of widely used and standardized languages for type descriptions, for example CORBA-IDL, ASN.1, LOTOS, GDMO and SDL, which fulfill some of the requirements of type descriptions in RM-ODP. It is not anticipated that any one existing language will address all of the needs of this standard, however, some may be adopted for description of particular ODP concepts. We choose OCL for the description of types for many reasons:

- Current trend in software engineering technology is the unification of methods which necessitates unification and integration of basic concepts and graphical notation. This is the objective of the UML language for the object-oriented development. The UML is formally being defined using the OCL language. In this context, we think that OCL will serve as a common denominator for formal method semantics and software engineering method semantics.

- OCL is object-oriented.

- The disadvantage of traditional formal languages is that they are usable to persons with a string mathematical background, but difficult for the average business or system modeler to use. OCL has been developed to fill this gap.

- OCL can be used to specify invariants on classes and types in the class model, to describe pre- and post conditions on operations and methods.

- The RM-ODP information viewpoint specification is described in terms of an invariant schema, a static schema and a dynamic schema which can be interpreted as follows: an invariant schema is the specification of the types of one or more information objects that will always be satisfied whatever behaviour the objects may exhibit. A static schema is the specification of the state of one or more information objects at some particular point in time. These types are subtypes of one or more of the types specified in the invariant schema. Behaviour in an information specification can be modeled as transitions from one static schema to another that is reclassification of instances from one type to another.

We deduce that OCL can be used to describe the information object types and, hence to describe the ODP informational specifications.

5.0 THE INFORMATIONAL SPECIFICATION OF THE ODP TRADING FUNCTION

5.1 Overview

The ODP functions specify the functionality of the execution environment for the ODP systems. The ODP execution environment masks the complexities inherent to the distribution and the openness ensuring several kinds of transparencies. The trading function is an ODP function that allows to realise other ODP functions. It is based on a database management function.

A detailed analysis of the ODP functions shows that the programming languages and the database management systems must be integrated. That is why we choose the ODP trading function as an application of our concrete information object model.

As the ODP trading is very detailed in [34], we describe here only its aspects, which are relevant to our study.

A definition of trading [34, 35] is as follows: “the activity of choosing services, such that they match some service requirements”. The choice is based on the comparison of the specification of a service (provided by a prospective consumer) and the service specification supplied by service providers or their agents. Trading is based on the notion of matching service offers and service requests. A service is a function provided by a component at a computational interface. The component responsible for the maintenance of the trading space and the matching of offers and requests is called a trader.

Current traders organise services in a service type hierarchy. Each service type defines an interface that prescribes the operations available for the interaction between the service provider and the service consumer. They also allow the association of a number of properties as attribute value pairs with each service type.

Central to the service type matching is the notion of type conformance. Type conformance is determined by the interaction interface. The specification of interfaces is then crucial and currently IDL-based.

Offering a service is called export, discovering a service is called import. To export, an object gives a trader a description of a service together with the location of a computational interface at which that service is available. To import, an object asks the trader a service having some characteristics, the trader checks against the descriptions of services and responds to the importer with the location of the selected service interfaces.

Due to the sheer number of service offers that will be offered worldwide, it is inevitable that the trading service will be split up and the service offered will be partitioned. Hence, the trading system consists of a collection of inter working linked traders, each of them manages a partition of service offers.

5.2 Informational Specification

The information of the trading system is a composite object described by the template, we call Information:

Information
Invariant schema : offers : Collection(Offer) nodes : Collection(Node) edges : Collection(Edge) partitions : Collection(Partition) *Initial schema* : Information() { offers = {}; nodes = {}; edges = {} partitions ={} } *Dynamic schema* : Export() : adds a service offer to service offer space of the trading system. Withdraw() : withdraws a service offer from the service offer space of the trading system. ModifyOffer() : changes the service property and service offer property values associated with a service offer whilst preserving the service offer identifier. AddEdge() : adds an edge to the trading system's set of edges. RemoveEdge() : removes an edge from the trading system's set of edges. ModifyEdge() : changes the property of an edge. AddNode() : adds a node to the trading system's set of nodes. RemoveNode() : removes a node from the trading system's set of nodes. Import() : searches for the subset of service offers which satisfy some matching criteria, scoping criteria and some preference constraints.

The component objects of the information object of the system are described as follows:

Offer
ServiceDescription : Service ServiceOfferIdentifier : Reference(Offer) ComputingInterfaceIdentifier : Reference(Interface) OfferProperties : Collection(Property)

Servicer
ServiceSignature : Collection(Operation) ServiceProperties : Collection(Property)

Partition
NodeRef : Reference(Node) NodeOffers : Collection(Offer)

Edge
FirstNode : Reference(Node SecondNode : Reference(Node) EdgeProperties : Collection(Property)

Node
NodeRef : Reference(Node) NodeProperties : Collection(Property)

The semantics of the information processing of the trading system is described in terms of pre- and post-conditions of each operation of the system. All the pre- and post-conditions are given in the context of an instance of the information template of the system (self).

Export(in NewOffer : Offer, in Anode : Node, out OfferRef: Reference(Offer))

Pre :
(1) self.nodes → includes(Anode)
(2) self.offers → forAll(p/ p.ServiceOfferIdentifier <> OfferRef).
(1) self.offers → includes(NewOffer)
(2) self.partitions → select(p/p.NodeRef=Anode.NodeRef).NodeOffers → includes(NewOffer).
(3) self.partitions → forAll(p,q / p.NodeRef <> q.NodeRef implies p.NodeOffers → Intersection(q.NodeOffers) → isEmpty).

WithdrawOffer(in OfferRef : Reference(Offer))
Pre :
self.offers → Exists(p/ p.ServiceOfferIdentifier = OfferRef)
Post :

(1) self.offers → forAll(p / p.ServiceOfferIdentifier <> OfferRef)
(2) self.partitions → forAll(p/ p.NodeOffers → Not Exists(q/ q.ServiceOfferIdentifier = OfferRef))

ModifyOffer(in OfferRef:Reference(Offer), ServiceProperties, OfferProperties:Collection(property))

Pre :
self.offers →Exists(p/p.ServiceOfferIdentifier = OfferRef)

Post :
(1) self.offers → Exists(p/p.ServiceOfferIdentifier = OfferRef and p.ServiceDescription.ServiceProperties = ServiceProperties and p.OfferProperties=OfferProperties)
(2) self.partitions → Exists(p/p.NodeOffers → select(q/ q.ServiceOfferIdentifier = OfferRef and q.ServiceDescription.ServiceProperties = ServiceProperties and q.OfferProperties = ServiceOfferProperties))

AddEdge(in NodeRef1,NodeRef2:Reference(Node); in EdgeProperties:Collection(Property))

Pre :
(1) self.nodes → Exists(p/ p.NodeRef=NodeRef1)
(2) self.nodes → Exists(p/ p.NodeRef=NodeRef2)
(3) self.edges → Not Exists((FirstNode=NodeRef1 and SecondNode=NodeRef2) or (FirstNode=NodeRef2 and SecondNode=NodeRef1))

Post :
self.edges → Exists(p/((p.FirstNode=NodeRef1 and p.SecondNode=NodeRef2) or (p.FirstNode=NodeRef2 and p.SecondNode=NodeRef1)) and p.EdgeProperties=EdgeProperties)

RemoveEdge (in NodeRef1, NodeRef2 : Reference(Node))

Pre :
(1) self.nodes → Exists(p/ p.NodeRef=NodeRef1)
(2) self.nodes → Exists(p/ p.NodeRef=NodeRef2)
(3) self.edges → Exists(p/((p.FirstNode=NodeRef1 and p.SecondNode=NodeRef2) or (p.FirstNode=NodeRef2 and p.SecondNode=NodeRef1)))

Post :
self.edges → Not Exists(p/ ((p.FirstNode=NodeRef1 and p.SecondNode=NodeRef2) or (p.FirstNode=NodeRef2 and p.SecondNode=NodeRef1)))

ModifyEdge(in NodeRef1,NodeRef2 :Reference(Node);in EdgeProperties : Collection(Property))

Pre :
self.edges → Exists(p/ ((p.FirstNode=NodeRef1 and p.SecondNode=NodeRef2) or (p.FirstNode=NodeRef2 and p.SecondNode=NodeRef1)))

Post :
self.edges → Exists(p/ ((p.FirstNode=NodeRef1 and p.SecondNode=NodeRef2) or (p.FirstNode=NodeRef2 and p.SecondNode=NodeRef1)) and p.EdgeProperties=EdgeProperties)

AddNode(in NodeRef : Reference(Node); in NodeProperties : Collection(Property))

Pre :
self.nodes → Not Exists(p/ p.NodeRef=NodeRef)

Post :
(1) self.nodes → Exists(p/p.NodeRef=NodeRef and p.NodeProperties = NodeProperties)
(2) self.partitions → Exists(p/p.NodeRef=NodeRef and p.NodeOffers → isEmpty())

RemoveNode(in NodeRef:Reference(Node))

Pre :
(1) self.nodes → Exists(p/p.NodeRef=NodeRef)
(2) self.partitions → Not Exists(p/p.NodeRef=NodeRef and Not(p.NodeOffers → isEmpty))
(3) self.Edges → forAll(p/p.FirstNode <> NodeRef and p.SecondNode <> NodeRef)

Post :
self.Nodes → Not Exists(p/p.NodeRef=NodeRef)

Import(in Aservice:Service; in MatchingCriteria, ScopeCriteria, refereneceCriteria:Expression; out Offers:Collection(Offer))

Post :
Offers = (self.offers → select(MatchingCriteria and ScopeCriteria and PreferenceCriteria)).

6.0 CONCLUSION

Now that the Reference Model for Open Distributed Processing has stabilised, attention is shifting towards the definition of ODP standards. The type repository function standard requires a model describing the types to be used in ODP systems. This would involve among others, determining what entities need to be typed and identifying (and characterising) language sufficient to describe the types identified. The ISO/ITU-T WG7 gives guidelines to achieve this objective, for example, the types required for the ODP functions and for the ODP viewpoint specifications should be considered.

Based on RM-ODP itself, we have defined a typing system. We have equally enumerated several advantages to use OCL for type descriptions. This work can be considered as a step to achieve the WG7 objective. We have used that typing system and OCL for the specification of the trading information viewpoint. This specification is simple than the ISO specification, which uses the Z language.

However, several areas require further work. One important issue is to complete the typing system by including the ODP engineering concepts. Also, we are investigating to what extent UML and OCL can be used as a formal notation for the development of ODP systems.

REFERENCES

[1] OMG, “The Object Management Architecture (OMA)”. Technical Report, December 1991, www.omg.com.

[2] OMG, “The Common Object Request Broker Architecture (CORBA), Architecture and Specification”. Revision 2.0 July 1995, www.omg.com.

[3] ISO/ITU-T, “Basic of Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use”. ITU/TS X901-ISO 10746-1, January 1995.

[4] ISO/ITU-T, “RM-ODP, Part 2 : Description Model”. ITU/TS X902-ISO 10746-2, January 1995.

[5] ISO/ITU-T, “RM-ODP, Part 3: Prescription Model”. ITU/TS X903-ISO 10746-3, January 1995.

[6] ISO/ITU-T, “RM-ODP, Part 4: Architectural Semantics”. ITU/TS X904-ISO 10746-4, January 1995.

[7] TINA-C, “Telecommunication Information Network Architecture”, www.cygnet.co.ul/TinaC/.

[8] J. P. Gaspoz, “Methodology for the Development of Distributed Telecommunication Services”, Journal of Software and Systems, June 1996.

[9] B. El Ouahidi and M. Bouhdadi, “Metodology for the Development of Distributed Sy stem”. JDIR’98 Conf. Paris, April 1998.

[10] B. El Ouahidi and M. Bouhdadi, “How to Develop a Telecommunication Application”, Telcom'97, Fes Morocco, 1997.

[11] G. Booch and al, “The Unified Modelling Language”, A Reference Manual. Addison Wesley, 1998. www.omg.com.

[12] UML-ODP, http://enterprise.shl/.com/uml-odp/uml-odp.html.

[13] E. Lupu and al, “A Policy Based Role Object Model”, First International Enterprise Distributed Object Computing Conference EDOC'97, Gold Coast, Australia, 1997.

[14] J. O. Aagedal and al, “Enterprise Modelling and QoS for Command and Control Systems”, Second International Enterprise Distributed Object Computing Conference (EDOC '98), San Diego, CA, USA, 1998.

[15] P. Linington and al, "Policies in Communities: Extending the ODP Enterprise Viewpoint", EDOC'98, San Diego, CA, USA, 1998.

[16] Z. Milosevic, and al, "Towards New ODP Enterprise Language", IFIP/IEEE Open Distributed Processing and Distributed Plateforms, 1997.

[17] ISO, http://enterprise.shl.com/other.

[18] B. Rumpe, "A Note on Semantics (with an Emphasis on UML)". Proceedings Second ECOOP, 1998.

[19] Ruth Breu and al, "Towards a Formalization of the Unified Modelling Language". In Proceedings of ECOOP'97. Springer Verlag, NCS, 1997.

[20] Ruth Breu and al, "Towards a Precise Semantics for Object-Oriented Modelling Techniques", ECOOP'97, Springer Verlag: NCS 1357, 1999.

[21] A. Evans and al, "Developing the UML as a Formal Modelling Notation", UML'98 Beyond the Notation, Ecole Superieure Mulhouse, Universite de Haute-Alsace, 1998.

[22] pUML, www.cs.york.ac.uk/puml.

[23] J. Warner and al., "Object Constraint Language OCL", Addison Wesley, October 1998.

[24] C. Hamilton and al, "JBDC Database Access with Java". JavaSoft Press, Addison Wesley, July 1998.

[25] J. M. Spirey, "The Z Notation, Reference Manual", International Series in Computer Sciences, Prentice-Hall International, 1998.

[26] C. B. Jones, "Systematic Software Development Using VDM". Prentice-Hall International, NY, 1988.

[27] C. A. R. Hoare, "Communication Sequential Process". Prentice-Hall International, 1998.

[28] A. Milner, "A Calculus of Communications Systems". Computer Sciences Spring-Verlag, 1985.

[29] D. Harel, "Statecharts, a Visaul Formalism for Complex Systems". Technical Report, Weizmann Institute of Sciences, Rehorot Isreal, February 1998.

[30] CCITT, "Specification and Description Language SDL" Technical Report, CCIT Rec. Z. 100, March 1988.

[31] ISO/ITUT-T, "LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavioral". ISO 8807, August 1988.

[32] CCITT, "Specification and Description Language 92, SDL 92". CCITT Rec. Z100, COMX-R 17 E, March 1992.

[33] ISO/ITU-T, "OSI-Part 4 Guidelines of the Definition of Management Objects". September 1991.

[34] ISO/ITU-T, "ODP Trading Function", ISO, Draft Rec. X.9tr, June 1995.

[35] OMG, "Trading Object Service Specification", in CORBAservices: Common Object Services Specification, December 1997.

BIOGRAPHY

Bouabid EL Ouahidi Obtained a PhD in Computer Sciences from the University of Caen at France. His current interests include developing specification and design techniques for use within Intelligent Network and TINA applications.

Mohamed Bouhdadi obtained a PhD in Computer Sciences from the Mohamed V University at Morocco. His current interests include developing specification and design techniques for use within Open Distributed System.