ELSEVIER

Computer Standards & Interfaces 19 (1998) 361–374

COMPUTER STANDARDS & INTERFACES

Common domain objects in the RM-ODP viewpoints

Guy Genilloud *

Swiss Federal Institute of Technology of Lausanne (EPFL) EPFL-DI-LIT, CH-1015, Lausanne, Switzerland

Abstract

An important standardisation effort is in progress within the Object Management Group (OMG) regarding domain computing facilities and common domain objects. In this paper, we investigate the very idea of common objects within the Reference Model for Open Distributed Processing (RM-ODP). We show that 'common objects' are in fact *common object templates*, that different kinds of templates are needed for different viewpoint models, and that agreeing on common *object templates* is particularly useful for information modelling.

Keywords: Object modelling; Object-oriented software engineering; Open Distributed Processing (ODP); Viewpoint model; Common domain objects; Common business objects; Standardisation

1. Introduction

Large companies in business domains such as manufacturing, healthcare, insurance, and finance, are currently addressing an important but daunting task: the standardisation of *domain computing facilities* that are specific to business domains, and the definition of architectures within which these facilities can be successfully developed and used. This standardisation implies agreements about objects that are common to a business domain, or even across multiple business domains—we will refer to all these objects as 'common domain objects'.

Agreements must first be found at an abstract (independent of implementation) and semantical level, prior to an implementation level. Otherwise, the use of common objects may result in an increased level of confusion, rather than in the desired improvements in commonness, complexity and reuse. However, in the absence of a reference implementation architecture and a common software engineering process, there is considerable difficulty in agreeing on just what an object is.

We argue that the Reference Model for Open Distributed Processing (RM-ODP) is a suitable basis for addressing the problems mentioned above. The RM-ODP includes a rich set of modelling concepts that are applicable to all of large distributed systems [16]. However, the RM-ODP is difficult to apprehend at first glance, and it is sometimes excessively general (being terse and overly open is often the price to pay for reaching international consensus, and for obeying ISO rules). As a result, its significance and usefulness are only beginning to be perceived by domain system designers.

We analyse in this paper the idea of common domain objects within the framework provided by the RM-ODP. Section 2 provides a general introduction to the RM-ODP, and concludes that common

* E-mail: guy.genilloud@di.epfl.ch

0920-5489/98/$ - see front matter
PII: S0920-5489(98)00023-3

domain objects can and should be considered for different ODP viewpoints, the enterprise, information and computational viewpoints being probably the most important. In Section 3, we investigate the information and the computational viewpoints in more detail, and show that it is preferable to begin by defining common domain objects in the information viewpoint. We conclude with recommendations for defining common domain objects.

2. Concepts and terminology for object-based modelling

There is still a fair amount of fuzziness and confusion around the concept of common domain object. Part of the problem is linked to the use of improper terms, as well as to insufficient qualification of the context in which the terms are used.

- By use of improper terms, we refer to the fact that the term 'object' is often used when the terms 'object type', 'object class', 'object template', or yet 'interface' would be more appropriate—e.g., people use the term 'common domain object' when the term 'common domain class' would seem more appropriate.

This problem is partially due to the fact that different computer science communities (information analysis, database, programming languages, operating systems), and even subcommunities within those communities, have developed their own perspective of object orientation. These perspectives are incomplete, and can be inconsistent with one another.

Another problem is that the usual 'object-oriented terminology' has drifted away from English. This leads to clashes between the 'object-oriented terms' and the English terms, which remain necessary and which are indeed used. For example, most languages or methods define 'class' as 'an object template from which instances might be instantiated.' This definition matches none of the uses of the word 'class' in English. This problem alone may explain the reluctance to speak of common business classes.

- By insufficient qualification of the context, we mean that it is not clear in what kinds of models the objects and related concepts are supposed to be used, and for what purpose.

Software engineering tells us that several models, at different levels of abstraction, are necessary to address the complexity of building large systems: an analysis model, a design model, an implementation model, and perhaps still other models. Accordingly, it is quite useful to qualify objects as '*analysis objects*' or '*implementation objects*', because these objects are not quite the same, as there need not be a one to one correspondence between analysis objects and implementation objects. However, this practice refers to a software engineering process and to an implementation architecture, and we have no universal agreement on such a process and architecture.

The RM-ODP responds to the first problem with a Foundations document [15]: it includes a rich set of modelling concepts, and it provides definitions that are both precise and mutually consistent. Importantly, the definitions are applicable to all kinds of object-based models, and they are independent of a notation, software engineering process, or implementation.

The RM-ODP solves the second problem by defining a set of five viewpoint languages (enterprise, information, computational, engineering and technology) that are a sufficient basis for addressing the modelling of large distributed systems [16]. It is important to note that ODP defines these languages independently of any software engineering process: the semantics of information and computational models, for example, are explained without any direct or indirect reference to an implementation.

2.1. The RM-ODP foundations

From the very beginning, ISO and ITU experts agreed that object-orientation concepts would be used heavily for specifying and building distributed systems. However, they immediately faced the problem that each of the different communities involved (information analysis, database, programming languages, operating systems) has its own perspective of object orientation.

To avoid misunderstandings, the RM-ODP Foundations document provides a rigorous definition for each of the concepts commonly encountered in object-oriented models [15]. These concepts have been successfully refined to serve each of the five ODP viewpoints, and new viewpoint-specific concepts are

defined using the RM-ODP Foundations concepts. The ODP Foundations thus represent the very invariant that has been applied equally well to enterprise modelling, information modelling, computational modelling, etc.—because of their generality of application and their precision, they capture the essence of what it means for a model to be object-oriented.

The RM-ODP Foundations introduce a general object-based model. This model comprises the following characteristics.

- An ODP system can be described (modelled) as a collection of related, interacting *objects* [14].
- *Action* is the most fundamental concept for modelling systems—it models 'something which happens'. ODP actions may have a duration and may overlap in time (allowing an action to model the exchange of a multimedia stream, e.g., a composite television signal).

All actions are associated with at least one object: *internal actions* are associated with a single object; *interactions* are actions associated with several objects.

- *Objects* are the units of encapsulation, characterized by their behaviour and their state. *Encapsulation* means that changes in an object state can only occur as a result of the internal actions or the interactions of that object. Objects have an *identity*, which means that each object is distinct from any other object.
- A *name* is a term which, in a given naming context, refers to an entity. In ODP, all names are relative to some naming context, and objects generally ignore the names that denote them. Moreover, there is no implication that all objects have a unique name in some 'well-known' naming context—deciding whether two names denote two different objects or the same object can be difficult or even impossible. [1]
- The *behaviour* of an object is a collection of actions with a set of constraints on when they may occur. Examples of constraints include sequentiality, non-determinism, concurrency or real-time constraints.
- An *interface* is an abstraction of the behaviour of an object. It consists of a subset of the interactions of that object together with a set of constraints on when they may occur.

In contrast with other object models, an ODP object can have multiple interfaces (the capsule of an ODP object is a set of interfaces). Like objects, interfaces can be instantiated and deleted (some objects have extendable capsules).

- A *template* specifies common features in sufficient detail to enable *instantiations* (this concept is usually called 'class' in the literature on object orientation). Thus, an object template is a specification of the common features of a set of objects; an interface template is a specification of the common features of a set of interfaces; an action template is a specification of the common features of a set of actions.
- A *type* is a predicate which classifies entities (objects, interfaces, actions, relations) into categories (therefore enabling reasoning about those categories). Entities can be typed with any predicate. Thus, an object or an interface can satisfy more than one type, and they can satisfy different types at different times.

The ODP notion of type addresses the true goals of typing, the ability to talk about, reason about and verify properties of things, and it is clearly independent of implementation. Commonly, entities are statically typed on the basis of the templates of which they are *instances*—a predicate, called a *template type*, is associated with an object template or an interface template. [2] However, it is possible to extend the benefits of typing to non-permanent properties of interest by explicitly defining 'dynamic types' (e.g., an hotel room may be 'free', 'reserved', or 'occupied') [8].

- An *object class*, in the ODP meaning, represents the set of objects that satisfy a given type. Many object models do not clearly distinguish between a specification for an object and the set of

[1] Fundamentally, object identity only implies that there exists a reliable way to refer to objects in a model. For example, an object can make reference to another object by associating a *reliable name* with it, in a private naming context—this name will then denote that very object until the referencing object performs another naming action with it. This is analog to the notion of *referential integrity* in CORBA [3].

[2] A template type characterises the template instantiations, usually by describing their suitability for some purpose. Objects and interfaces need not be instantiations of a given template to be instances of its template type.

objects that fit the specification. ODP makes the distinction between template and class explicit.

A *subclass* is a subset of a class. A subtype is therefore a predicate that defines a subclass. ODP subtype and subclass hierarchies are thus completely isomorphic.

- *Abstraction* is the process of suppressing irrelevant detail to establish a simplified model, or the result of that process. Composition, viewpoint (Section 2.2), and view (Section 3.1.3) are three distinct techniques of abstraction.
- *Composition* is the combination of two or more entities (objects, behaviours, actions) yielding a new entity of the same kind, at a different level of abstraction. The characteristics of the new entity are determined by the entities being combined and by the way they are combined.

For example, two actions may be composed by sequential composition, yielding a new (composite) action at a different level of abstraction. Abstraction is obtained in the sense that one needs only consider the pre- and the post-conditions of the composite action, and not those of its consisting actions.

The ODP Foundations define many other important concepts, such as *activity*, *domain*, *epoch*, *fault*, *policy*, *name*, and *role*. We invite the reader to consult the standard documents [14] and [15] for the definitions and a discussion on those concepts.

2.1.1. Discussion

An important characteristic of the ODP Foundations object model is that it is very general and that it makes a minimum number of assumptions. For instance, objects can be of an arbitrary granularity (e.g., they can be as large as the telephone network, or as small as an integer); objects can exhibit arbitrary (encapsulated) behaviours, and have an arbitrary level of internal parallelism.

The reader should note that, according to the ODP Foundations, encapsulation is a property specific to objects, whereas inheritance (template derivation) and classification (typing and subtyping) are not. Indeed, template derivation and typing are applicable not only to objects, but also to interfaces, actions, etc.

Encapsulation ensures that any change in the state of an object can only occur as a result of an action of that object (i.e., state changes are under an object's control). It is then possible to define abstractions for an object, for example invariants that characterize its state. In some modelling techniques, an object's interfaces need not be specified explicitly. Encapsulation is then provided by defining specific invariants for the object, and by not letting these invariants be overruled. [3]

Finally, note that interactions between objects are not limited to message passing. They may include, for example, asynchronous and multiway synchronous interactions. *Asynchronous interactions* are perceived by objects at different times, as in the sending and receiving of a letter. *Synchronous interactions* provide a rendezvous mechanism between objects: objects participate jointly in such an action, and they may change their states simultaneously. A synchronous interaction is '*multiway*' when it involves more than two objects.

An action may be atomic or non-atomic at a given level of abstraction (i.e., within a given model). *Atomic actions* cannot be subdivided into other actions. *Non-atomic actions* are essentially a notational convenience. This paper is essentially concerned about atomic interactions between objects. Therefore, all the actions that we discuss are atomic. In particular, we will use the term '*multiway interaction*' to denote atomic multiway synchronous interactions.

2.1.2. Compatibility with classical and generalized object models

As explained by Kilov and Ross [9], there are two broad categories of object models.

- *Generalized object models* do not distinguish a recipient from other request parameters [13]. In generalized models, a request is defined as an event which identifies an operation and optional parameters. For example, an `AddToInventory (part1, lot2, bin3)` request does not give special significance to either the part, the lot, or the bin. These objects participate uniformly in the request.

[3] Invariants may be applied not only to a single object, but also to sets of objects. Obviously, encapsulation of a given object enforces only the invariants that are confined to that object.

● *Classical* or *messaging object models* do distinguish a recipient. In classical models, a request is defined as an event which identifies an operation, a recipient, and optional parameters. Either the part, the lot, or the bin could be designed to be the recipient of the AddToInventory request. The request to a specific recipient is called a message. A common syntax places the recipient first: `part1. AddToInventory (lot2, bin3)`.

The RM-ODP Foundations are compatible with generalized object models in the sense that multiway interactions correspond to '*generalized requests*.' However, the concept of multiway interaction is that of an action occurrence, not that of an action request. A multiway interaction simply does not occur when one of the objects involved is in a state that is incompatible with it; a generalized request always occurs, but it may return an error if its preconditions are not satisfied. The generalized object model semantics can be emulated in ODP by specifying a choice between several multiway interactions.

The RM-ODP Foundations are compatible with classical object models in the sense that interactions can be constrained, as does the ODP computational language (see Section 3.2).

2.2. The five ODP viewpoints

Distributed systems are complex systems, and they usually involve a number of concerns, ranging from business issues (e.g., can the customer credit limit policy be overruled in some circumstances?) to technology issues (e.g., does the Java language offer sufficient performance for computing a particular function?). To deal with this complexity, the RM-ODP applies the principle of separation of concerns, and introduces five viewpoints. [4]

● The *enterprise viewpoint* defines and explains the objectives, the scope and the responsibilities of a system. For this, it is necessary to understand the overall objectives and responsibilities of the *communities* (configurations of enterprise objects) that are using the system, and of other communities that have a stake in it. Objectives and responsibilities (and thus, behaviour) are specified by a *contract*. The system typically appears as one or several enterprise objects playing a number of *roles*: its behaviour is specified in terms of objectives and policies, and interactions represent transfers or changes of responsibility. Enterprise modelling is important for understanding and deciding the requirements placed on the system, and for recording the motivation behind a system specification.

● The *information viewpoint* focuses on the *semantics* of the information that is held or exchanged by the system, and on the processing of that information. The result is a system specification that is easily comprehensible: ideally, the important properties of the system are stated explicitly in a declarative way, rather than implemented (say 'what', not 'how'). In some sense, an information specification is independent of how a system is built. However, it is often necessary to revise an information specification is that a distributed implementation is possible, or so that legacy systems can be reused effectively.

● The *computational viewpoint* focuses on the *components* of a distributed system. Computational objects are loosely coupled components, with well-defined interfaces, that can be built independently and that can be distributed over a network. Distribution is only considered to the extent that computational objects are specified in a manner where they can be distributed, rather than how they are distributed; it is not necessary to specify the location of an object, nor whether it can migrate or not. Furthermore, resource usage (CPU, memory, communication) needs not be considered.

● The *engineering viewpoint* is concerned with the way the system is physically distributed and configured, taking into account the trade-off between processing capacity, communications bandwidth, transparency requirements (e.g., fault tolerance), and quality of service issues. For simplicity, semantic concerns need not be considered in the engineering viewpoint—the complex evaluation of business rules (the business logic), or other information processing activities, can be abstracted out and substituted by non-determinism and numeric estimates of resource usage.

[4] The RM-ODP Foundations define *viewpoint* as a form of abstraction achieved using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within a system. The RM-ODP Architecture selects and defines five viewpoints as a necessary and sufficient set for the needs of ODP.

● Finally, the *technology viewpoint* specifies the actual artefacts and technologies which make up the system. It provides the link between the set of specifications and the concrete implementation.

In summary, the adoption of a viewpoint is a technique for making system models that emphasize one particular concern, while ignoring other characteristics that are temporarily irrelevant to that concern.

2.2.1. Viewpoint models and system specification

In every viewpoint, the goal is to produce a viewpoint specification, i.e., an object-based model that represents a system and (some of) its environment. The full specification of the system is then the collection of all the viewpoint specifications, i.e., the system implementation must conform to all of those.

The RM-ODP defines five viewpoints, but this does not imply that exactly five models are always sufficient to specify and build a distributed system. In fact, it is not always necessary to specify a system in all the five viewpoints, or to make complete specifications in every viewpoint. Practitioners sometimes construct only elements of a viewpoint specification—they stop before completing a full object-based model. For example, only the 'static part' of the information specification is explicitly defined; the dynamic part is to be deduced from other viewpoint specifications. This practice is only possible in simple cases, and outsiders may find it confusing as they do not understand what has been specified in the viewpoint. However, this way of working with multiple viewpoints remains useful for complexity management, i.e., for separating the concerns.

On the other hand, there is often an interest in making more that one complete object-based model within a viewpoint. For example, it can be useful to make a computational model where a certain service is provided by a single computational object (to show how it will be used), and another model where the same service is provided by a composition of computational objects (to show how it will be distributed).

2.2.2. Viewpoint languages

The RM-ODP defines a viewpoint language for each viewpoint, defining concepts and rules for specifying systems from the corresponding viewpoint. The ODP viewpoint languages are all object-oriented, and they each specialise the RM-ODP Foundations by refining the fundamental concepts, by introducing prescriptive rules, and by introducing viewpoint-specific concepts (defined in terms of the fundamental concepts). In short, each viewpoint language supports the creation of object-based models that address the concerns relevant to the corresponding viewpoint.

It is important to note that the RM-ODP uses the term '*language*' in its broadest sense: ''a set of terms and the rules for the construction of statements from the terms''. The RM-ODP does not propose any *notation* for supporting the viewpoint languages —notations are the scope of specific ODP standards. [5]

2.2.3. Correspondences between ODP viewpoint models

Different viewpoint specifications are *not* different views or different abstractions of a same object-based model. A (complete) viewpoint specification defines, at some level of abstraction, a complete object-based model, e.g., an information and a computational specifications define two different (but related) object-based models. More to the point, ODP objects need not be part of models in different viewpoints. In fact, they cannot—the objects in an ODP viewpoint model are only defined with respect to that model. For this reason, objects in enterprise models are called *enterprise objects*, objects in information models are called *information objects*, etc.

Strong correspondences are possible between models in different viewpoints, but ODP defines few rules regarding these correspondences. For example, there need not be one-to-one relationships between objects in the models. Unlike software engineering processes, ODP avoids providing design heuristics and specifying arbitrary consistency rules. As observed by R.A. Tyndale-Biscoe, ''the relationships

[5] However, ISO and ITU are working on an amendment to the RM-ODP Part 4, which will explain how to use existing formal specification languages (LOTOS, Z, SDL and Estelle) for producing models in the ODP viewpoint. This work does not imply that the RM-ODP mandates the use of formal description techniques for ODP systems.

(correspondences) between (two) different ODP viewpoint specifications are driven by, and only by, the fact that both specifications describe the same real world thing, the system (the ODP specifications are of the system—they describe a thing in the real world that we call the system) [12].''

Defining non-arbitrary correspondence rules between different viewpoint specifications is a very difficult problem in general. However, specific correspondences are easier to establish in practice, i.e., with respect to a specific system. Of course, specifiers must record those correspondences whenever they define or discover them. Correspondences between viewpoint models arise also naturally because, in the words of R.A. Tyndale-Biscoe, ''objects in ODP viewpoint specifications are related to (rather than describe) other things in the real world that are of interest in some way to the business being supported (and this is what 'business objects' are all about). Thus, an information object is related to some class of thing in the real world about which the system has to know something. And we may have computational objects that are related to real world things.''

2.3. Common domain objects in the RM-ODP framework

Following the ODP terminology, it seems more appropriate to speak of '*common object templates*' than of 'common objects'. Indeed, the idea is not to have different companies or different systems make use of the very same object instances, but to increase the commonalities between different object-based models. This goal can be achieved by using common object templates.

In fact, defining and agreeing about '*common object types*' becomes necessary as soon as common object templates are defined. The main reason is that objects do not exist in isolation—they are always related to other objects. In such 'object relations', whatever they are, it is the type (or properties) of the object that is of interest, not all the characteristics of its instantiation template. Considering the type, rather than the template, is indeed essential for enabling subtyping and substitutability.

Whether we want it or not, a template type is associated with every template that we define and use. Defining this type explicitly is much preferable. [6] This can be done in two ways: by well-defined typing rules which map templates to template types, or via an explicit and separate definition of types. The first solution is the most attractive but it is not sufficient by itself—objects may indeed have types which have little to do with templates and instantiation (see for example the concept of dynamic type in Section 2.1).

In summary, we observe: that the so-called 'common objects' are in fact common object templates or common object types, that common object templates cannot be defined in isolation (rather, they need to be defined in related groups or sets), and that agreeing on common types is probably more important than agreeing on common templates. Nevertheless, in this paper, we mainly discuss common domain object templates rather than types, because these are the 'objects' that people look for and want to (re)use. Moreover, it is probably easier to define and to agree on common types if common templates are also defined.

In this paper, we are specifically concerned with the characterization of common domain object templates in the information and the computational viewpoints. We do not discuss the engineering and technology viewpoints any further because they are not concerned with domain semantics. We recognize that defining common domain object templates (or role templates) is probably a pertinent idea for enterprise modelling, but we do not consider this topic in this paper. A group of experts within ISO is currently refining the RM-ODP enterprise viewpoint language—we await more results from this work before completing our research in this viewpoint.

3. Common objects in the computational and the information viewpoints

In the second part of this paper, we show that common templates for information objects should be

[6] Template types are not defined with respect to interface templates in OMG IDL. As a result, the meaning of types and subtypes in CORBA has been the subject of many discussions, with no clear resolution. The current modus operandi is that subtyping rules are defined indirectly via a CORBA communication protocol specification (known as GIOP).

different from those for computational objects, and that it is preferable to begin with definitions of common information object templates. First, we develop our introductions to the information and the computational viewpoints, and we discuss the relation between them.

3.1. The ODP information viewpoint

Not much is said about information modelling in the RM-ODP standard. This is quite right for a reference model on distributed processing, but the intent of information modelling is perhaps insufficiently defined: the statement that ''an information specification defines the semantics of information and the semantics of information processing in an ODP system'' can be interpreted in different ways.

A useful interpretation can be obtained by noting that ODP defines *information* as ''the knowledge that is exchangeable amongst users in a given universe of discourse.'' In this context, information may be seen as ''the knowledge necessary to make use of a system or part of the system'' [2]. An information model thus specifies what the totality of users must know about a system, to make a proper use of it (see also Section 3.1.1). Note that this implies an understanding of the information that is to be provided to the system, and of the information that is obtained from it. Note also that a user of a system may be a person or another system—in the latter case, the users of the information model are the people who specify, design or implement that other system.

Simplicity and independence of implementation are major concerns of information modelling, whilst executability of models is not. Users need not be aware of all the components of a system, of all the possible states of these components, nor of all the interactions between these components. They are only interested in the system states that they can perceive. Therefore, a goal of information modelling is to produce a system specification that includes only those states, or as few more as possible.

In summary, information modelling tends to specify a system by using a minimal number of conceptual states, and by using concepts that make sense to the users of the system.

3.1.1. The information language modelling technique

In the information viewpoint, like in any other viewpoint, a model consists of a configuration of interacting objects. This configuration of objects, the objects within it, and its overall behaviour, can be specified in different ways, that we call *modelling techniques*. Note that we do not address the issue of notation: a same modelling technique can be supported by different notations (and notations are clearly outside the scope of the RM-ODP).

The RM-ODP information language is not very prescriptive regarding the modelling technique that is to be used in the information viewpoint. It simply proposes a modelling technique analog to Z, or the analysis phases of Refs. [1,6]. It does this with only a few rules and definitions, saying little more than the following:

- *Static schemas* capture information structure (global state) at some point in time.
- *Dynamic schemas* specify allowable state changes in multiple information objects, using pre-conditions and post-conditions.
- *Invariant schemas* constrain the possible states and state changes of the objects to which they apply.
- A state change involving a set of objects can be regarded as an interaction between those objects. Not all of the objects involved in the interaction need change state; some of the objects may be involved in a read-only manner.

The above rules raise two important questions:

1. How should information models be expressed in notations that do not provide either dynamic nor invariant schemas (e.g., LOTOS)?
2. How can dynamic schemas and invariant schemas specify legible state changes in several objects, when encapsulation is an essential property of objects (as noted in the ODP Foundations)?

An answer to both questions is that 'multiway interactions are allowed in information models.' Multiway interactions are a powerful modelling concept for two reasons. Firstly, a single multiway interaction can consistently change the states of several objects, which is often what is desired. And secondly, a multiway interaction occurs and produces the right effects, or it simply does not occur at all. Therefore, multiway interactions always leave the system in a well-defined state.

By themselves, multiway interactions allow information models to be more abstract and simpler to understand than classical object models, such as computational, engineering, 'design' or 'implementation' models. However, the information language allows for an even greater level of abstraction by not requiring interactions to be listed explicitly; an information object may participate in all interactions that one can think of, except those that conflict with the constraints of dynamic and invariant schemas. In other words, information modelling proceeds by excluding invalid interactions (adding constraints), rather than by listing valid interactions. Typically, this is done by specifying invariants (invariant schemas) that pertain to the states of several objects.

3.1.2. Interactions between the system and its environment

To understand a system specification, it is necessary to know the hypotheses made about its environment (otherwise, there are simply too many potential interactions and state changes to consider). This can be done by specifying the interfaces of the system, i.e., by specifying explicitly all the valid interactions of the system and its environment. This approach is consistent with the fact that a system is accessed via specific interfaces, and that users need to know these interfaces to make use of the system.

However, it is possible to abstract over the interactions of the system by constraining the states of the system with the states of its environment. This is done using invariant and dynamic schemas that pertain both to information objects in the system, and information objects in its environment. It can be interesting to use this technique at a high level of abstraction, or when providing a specific 'information view' of the system.

If an information specification includes neither the system interfaces nor information objects in the system environment, then something is missing in that viewpoint specification.

3.1.3. Multiple information views of a system

As a distributed system has typically different kinds of users, it can be interesting to specify several information models, or *information views* (see Ref. [1] for a good example of using multiple views in specification). Different views may be defined for different users, or even for the same user in different contexts.

Within information views, some interactions between the system and its environment may be abstracted as we explained above. For example, an information view may tell a clerk that, as far as he or she is concerned, a shipment's location information in the system always corresponds precisely to the effective current location of the shipment; an invariant schema constrains both the location objects inside and outside the system to have the same states at all times. A different information view will specify the system interactions that update a shipment's location information. A more thorough example is provided in Ref. [4], Section 4.5.1.

3.2. The ODP computational viewpoint

Unlike information objects, computational objects cannot participate in multiway interactions. This restriction underlies an important difference between information and computational modelling.

The fundamental idea of the computational viewpoint is that computational objects identify loosely coupled components and their interactions, such that an implementation with an 'engineering virtual machine' (e.g., CORBA) is possible without much interpretation. For this very reason, the computational language constrains the interactions in which objects may participate to signals, operations, and flows.

- *Signals* are synchronous interactions between two objects and they provide only one-way communication. As all synchronous interactions, they are always perceived identically by both participants in the interaction. Hence, there is no concept of partial failure of a signal.

Importantly, computational objects cannot, in general, communicate directly using signals. What objects can do is to instantiate a so-called *binding object* between them (a binding object is typically an abstraction of a communications protocol)—signals can then be exchanged with the binding object, which propagates them with a delay to the other objects.

- *Operations* are interactions analog to requests in CORBA, or to messages in object-oriented languages. They involve at most two objects, and they are classified in two types: *announcements* are oper-

ations for which no outcome is reported to the invoker; *interrogations* are operations for which an outcome (*termination*) is always reported to the invoker. This reported outcome may be the real termination of the operation, but it can also be an exception reporting an engineering infrastructure failure (e.g., a communication failure).

Operations are *asynchronous* in the sense that the invocation and termination might be delivered some time later (if at all) after they have been submitted. In fact, operations might sometimes be synchronous (submission and delivery are one and the same event), but the assumption to be made is that they are asynchronous. Note that we are not interested in whether an operation invocation is *blocking* (the thread waits for the termination) or *non-blocking* (the thread executes other actions before attempting to receive the termination). This is an implementation issue.

- *Flows* are interactions modelling the conveyance of information from a producer object to a consumer object. They are typically used for modelling continuous interactions including the special case of an analogue information flow. As for operations, the assumption to be made is that flows are asynchronous.

Interactions in computational models are constrained because implementing multiway interactions (including signals) in a distributed way is inefficient or excessively difficult—allowing arbitrary interactions would defeat the very purpose of the computational viewpoint. Restraining interactions is also a way to preserve a clear separation between computational modelling and information modelling, as discussed in Section 3.1.1.

In fact, some *ODP transparencies*, such as the *transaction transparency*, can relax the constraints imposed by the computational language in specific and controlled ways. We do not consider the impact of the ODP transparencies in this paper.

3.3. Relations between computational models and information models

It is often said in ODP tutorials that the ODP viewpoints do not correspond to levels of abstractions. It is sometimes stated that a computational model is not a refinement of an information model. These statements are certainly not wrong, but they need to be qualified.

3.3.1. Development process

Above all, it must be emphasised that the ODP viewpoints do not suggest a 'waterfall' development process. Rather, ODP modelling, whilst usually initiated in the enterprise viewpoint, is most successful if several viewpoints are considered in parallel (i.e., in an iterative way).

Consider, for example, the information, the computational, and the engineering viewpoints. Analysts strive for simplicity in information models—as a result, they tend to make information models that are too 'ideal' for an implementation to be possible, given the constraints imposed by distribution, performance objectives, and technology. Such feasibility problems will be fully realised when working on the computational and engineering models. When this happens, the information model must be amended to make it less constraining with respect to the implementation. This may be done in a direct way by relaxing some integrity constraints (invariants), or by introducing extra information objects in the model.

Assume for example that the stocks of a distant warehouse are modelled by a 'stocks' information object, and that system operations are defined with respect to that information object. This model may be impossible to implement because precise and timely information cannot be made available regarding the warehouse stocks, or because too many clients access the warehouse system concurrently. The information model can be made more flexible by the introduction of an extra information object, say a 'stocks estimator' object; system operations are then redefined in terms of the 'stocks estimator' object. The 'stocks' object remains in the model as it is useful for explaining the semantics of the 'stocks estimator' object. Note that such a change may affect the enterprise model.

3.3.2. Levels of abstraction

A computational model is more concerned with distribution and technology than an information model. It appears, therefore, to be less abstract than an information model. However, talking about levels of abstraction can be deceptive because there exist different kinds of abstractions. For example compu-

tational models can be made more or less abstract, as we explained. As another example, an enterprise specification can be detailed about actions and objects in the system environment, whilst the other viewpoint specifications typically abstract most aspects of the environment. Thus, the ODP concept of viewpoint is not that of a level of abstraction.

However, abstraction can be understood, among others, as the degree of dependence with respect to the system implementation. In this sense, and in this sense only, the ODP viewpoints can be related to levels of abstraction; enterprise being the most abstract and technology being the most concrete. Franklin and Robinson [5] used this idea in a thoughtful presentation at a recent workshop on ODP, which was well-received.

3.3.3. Refinement of an information specification

There is no refinement relationship between the information and the computational viewpoint, for at least three reasons. First, as we have seen, the development of a computational model may induce amendments to an information model. Second, viewpoints are not levels of abstraction. And third, an information model is not meant to be a first attempt at a computational model (ideally, an information model is fully independent of distribution).

Nevertheless, a computational model and an information model can be linked by a refinement relation. Indeed, Herbert [7], the ISO rapporteur for the RM-ODP Architecture, wrote recently:

In some sense, the computational language could be thought of as a terminating condition for refining an information specification to the point where all behaviour have been localised to computational interfaces and expressed in terms of the actions in a portability model (viz. a programming language).

It is conceivable to apply refinement steps to (i.e., to modify) an information specification until all the constraints of the computational language are satisfied. When this is done, the refined information specification can be considered (relabelled) as a computational specification, and used as such.

Note that there is no obligation, nor even a suggestion, to arrive at a computational specification by applying refinement steps to an information specification: the RM-ODP proposes no refinement process, nor any refinement steps or rules. That the behaviour of the system in the computational model (a configuration of computational objects) be behaviourally compatible with the specification of the system in the information model (a configuration of information objects) is what is required. In particular, all the system states in the information specification must have a related state in the computational specification.

3.3.4. Computational models used as information models

All computational models satisfy the rules of the information language. Any computational model may thus be considered and may even be used as an information model. Such a model is probably not the simplest nor the most abstract information model that can be made, but it represents a valid specification of a system, and users can extract enough knowledge from it to use the system.

Although in general we do not recommend this practice, it sometimes makes sense to use a computational model as an information model. For example, if there are just two components to consider, revealing the system components does not dramatically increase the complexity of the model. Moreover, revealing components in an information model can be useful when they tend to enter in well-known failure modes.

3.3.5. Specification of computational objects

The computational language does not constrain the behaviour of computational objects: this behaviour may be specified by any means deemed appropriate [10]. This is essential to enable computational objects to ‘encapsulate’ legacy systems, and even human users.

The behaviour of a computational object can be specified using the computational language alone, but this is cumbersome because it lacks expressive power. Another possibility is to consider the computational object as a system and to apply the viewpoints recursively to it; its behaviour is then specified in an information viewpoint model.

If this approach is followed, then the relation between the information specification and the computational specification can be made more visible; the information object templates used for specifying

the system can be reused in the specification of the computational objects. An interesting example was shown in the context of the LOTOS language, in a draft ISO technical report [11].

3.4. Common object templates

From the Sections 3.3.3 and 3.3.4, we understand that some object templates can be used in both computational models and information models. Nevertheless, information objects differ from computational objects in several ways:

1. Information objects are more conceptual because they model entities of interest to users, while computational objects represent units of implementation.
2. Information objects may be used in the specification of computational objects, while the opposite makes little sense.
3. Information objects can interact with each other in ways which are impossible for computational objects.

The second point above shows that it is preferable to begin with defining common templates for information domain objects. The third point indicates that information objects can be composed more easily and in many more ways than computational objects. Thus, common templates for information objects should be different from those for computational objects.

3.4.1. Encapsulation and behaviour of information objects

In the information viewpoint, encapsulation means that the behavioural constraints expressed in an information object template cannot be overridden nor violated—this is 'enforced' by the syntactical rules of the modelling notation, or simply by its semantics. Thus, an information object template consists of a specification of the possible states and state changes of an instance (typically, this is done using an invariant, or rules governing the transitions between those states), and operations need not be specified.

However, some operations may be specified: an object template may specify one or several interfaces explicitly, and still be an interesting information object template. There is nothing wrong with emphasizing certain interactions, provided other interesting and valid interactions are not excluded.

Even when no operations are specified for them, information objects are not simply 'lists of attributes.' They do have a behaviour, that must be specified. *Data* can be considered as special objects with no constraints in their states and state changes (beyond the constraints provided by the datatypes of their 'attributes'). Clearly, common domain information object templates should be defined as precisely as possible—they should be more than data 'templates.'

3.4.2. Encapsulation and behaviour of computational objects

In the computational viewpoint, encapsulation of an object means that all its interfaces are specified explicitly—all accesses and modifications to the state of an object are mediated by the interactions listed in those interfaces. Thus, a computational object template specifies all the operations, signals and flows of its instances; it specifies also the behaviour that occurs behind those interfaces.

The computational language does not constrain the behaviour of computational objects beyond the contracts imposed by their interfaces (e.g., an object which invokes an interrogation must accept to receive the termination of that operation—it cannot 'forget' to enquire about its results). One possibility for specifying the behaviour of a computational object is to use an information language, with all its power. Thus, the computational object is treated as a system, and we make an information specification for it; moreover, we strive to use common information object templates in this specification. If that approach is chosen as we expect, computational object templates will contain constraints, just as information object templates do. However, these constraints are not enforced by the computational language; they are to be enforced by the implementation of the computational object. [7]

3.4.3. Reuse of common object templates

The information language specification technique is not only very expressive, but it also makes infor-

[7] An implementation can enforce these invariants precisely because all the object interfaces are defined and enforced.

mation object templates (units of specification) easily reusable. As we explained, information modelling normally proceeds by excluding invalid interactions (adding constraints), rather than by listing valid interactions. Clearly, an object is more reusable if it can participate in more interactions.

For example, consider a multiway interaction that is shared by a clerk object, an inventory object and a customer shipment object; the occurrence of this interaction models that a clerk decides to ship some goods to a customer, that these goods are withdrawn from the inventory, and that they are added to the shipment. This multiway interaction occurs only if the three objects are in a state which allows it to happen.

Now, suppose that new requirements indicate that a shipment implies an immediate payment. The model can accommodate this change very easily by having two more objects participate in a revised multiway interaction: a customer account object and a vendor account object. Constraints are added in the dynamic and invariant schemas, but the templates of the clerk object, the inventory object, and the customer shipment object remain unchanged.

The above example illustrates the interest of information modelling for defining reusable common domain objects, or more specifically, common information domain object templates.

3.4.4. Event notifications

In the domain of telecommunications, *event notifications* (sometimes incorrectly called *events*) are interactions provided by an object for the benefit of other objects: an object is willing to notify other objects about actions occurring within itself; but there is no expectation that any other object will act on these notifications. This concept of event notification is interesting for common object templates, because events can be defined independently of whether other objects want to be notified of them—an object needs not know which other objects are interested in the events it emits.

In an information model, every change of state can be considered an event: an object can learn about a state change in another object, even though no event notification is explicitly associated with that change of state. By not defining event notifications explicitly, reuse is facilitated. In fact, there can also be an interest in defining an event notification if something of interest (an event) happens in an object, but results in no change of state.

The ODP computational language does not include explicitly an interaction concept of event notification. However, this concept can be introduced by the definition of a notification service. Event notifications are then realised by operations. Because it can be cumbersome to specify and implement the operations for emitting and subscribing to event notifications, a notation for object templates could introduce an explicit concept of event. Tools would then be used to generate the code for the notifications of those events.

It is important to note that the concept of event notification is more powerful (and therefore more useful) in an information model than it is in a computational model. In an information model, an event notification may become a part of a multiway interaction (notified objects change their state at the same time as the emitting object does). In a computational model, objects learn about events some time after they have occurred; moreover, the ordering of event notifications does not correspond to the temporal or causal ordering of the event occurrences. For that reason, event notifications in an information model should not necessarily be refined into event notifications in a computational model.

4. Summary and conclusion

We used the Reference Model for Open Distributed Processing (RM-ODP) as a basis for investigating the concept of common domain object. In the ODP terminology, we found that ‘common objects’ are in fact common object templates, or common object types. Common types are probably more important than common templates. Nevertheless, we attached ourselves to the concept of common object templates, because these are the ‘objects’ that people look for and want to (re)use. Moreover, it is probably easier to define and to agree on common types if common templates are defined as well.

We looked more closely at computational and information object templates, because they are particularly relevant to the standardisation of computing facilities. We showed that common templates for

information objects should be different from those for computational objects. In particular, interfaces and operations need not be specified for information objects, whilst they are essential for computational objects.

We also showed that it is preferable to begin with definitions of common information object templates. The latter are indeed more easily reusable, and they can be used for defining computational object templates, whilst the opposite makes little sense. This latter point means that agreeing on information objects might be considered as a prerequisite for agreeing on the semantics of computational objects, or in other words, for reaching a common understanding of components in domain facilities.

Acknowledgements

Jean-Bernard Stefani gave me some useful hints to understand the RM-ODP information model. My position and arguments in this paper were influenced and improved by E-mail discussions with Andrew Herbert, Haim Kilov and R. Alexander (Sandy) Tyndale-Biscoe. Rolf Eberhardt and Marc Zweiacker, of Swisscom, provided valuable advice and comments to an earlier version of this paper.

References

[1] D. D'Souza, A. Wills, Catalysis—practical rigor and refinement: extending OMT, fusion, and objectory, http://www.iconcomp.com / papers / catalysis / catalysis.frm.html, 1995.
[2] H. Christensen, E. Colban, Information modelling concepts, Technical Report, Telecommunications Information Networking Architecture Consortium (TINA-C), April 1995.
[3] OMG, The common object request broker: architecture and specification (2.0), Object Management Group, 1995.
[4] G. Genilloud, Towards a distributed architecture for systems management, PhD Thesis 1588, Computer Science, Swiss Federal Institute of Technology of Lausanne (EPFL), 1996.
[5] D. Franklin, P. Robinson, Putting the OMG's OMA on the map using RM-ODP: how do and how should the OMA and ODP relate?, Proceedings of the OMA-ODP Workshop, Cambridge, UK, 1997 (see also http://www.omg.org/omaodp/OMAP.htm).
[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, et al., Object-Oriented Development: The Fusion Method, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[7] A. Herbert, private communication, Nov. 17, 1997.
[8] W.H. Harrison, H. Kilov, H.L. Ossher, I. Simmonds, Technical note—from dynamic supertypes to subjects: a natural way to specify and develop systems, IBM Systems Journal 35 (2) (1996) 244–256.
[9] H. Kilov, J. Ross, Information modeling: an object-oriented approach, Prentice-Hall, Englewood Cliffs, NJ, 1994.
[10] E. Najm, J.-B. Stefani, A formal semantics for the ODP computational model, Computer Networks and ISDN Systems 27 (8) (1995) 1305–1329.
[11] ISO/IEC and ITU-T, Use of formal specification techniques for ODP, ISO Technical Report, First Working Draft, 1992.
[12] R. Tyndale-Biscoe, (private communication), Jan. 9, 1998.
[13] ANSI, The X3H7 object model features matrix, Technical Report X3H7-93-007v10, http://info.gte.com/ftp/doc/activities/x3h7/by_model/OODBTG.html, February 14, 1995.
[14] ISO/IEC and ITU-T, Open distributed processing—basic reference model: 1. Overview and guide to use, Standard 10746-1, Recommendation X.901, 1996.
[15] ISO/IEC and ITU-T, Open distributed processing—basic reference model: 2. Foundations, Standard 10746-2, Recommendation X.902, 1995.
[16] ISO/IEC and ITU-T, Open distributed processing—basic reference model: 3. Architecture, Standard 10746-3, Recommendation X.903, 1995.

Guy Genilloud holds a B.Sc. in Electrical Engineering (1981) and a Ph.D. in Computer Science (1996) from the Swiss Federal Institute of Technology of Lausanne (EPFL), and an M.Sc. in Computer Science (1987) from Queen's University, Kingston, Ontario. He is currently a senior researcher in the Computer Science department of EPFL, where he has worked since 1987. He has lead EPFL's participation in several industrial research projects, in particular the European projects ISA (Integrated Systems Architecture for ODP, ESPRIT 2267) and SysMan (Open Distributed Systems Management, ESPRIT 7026). Guy Genilloud has represented the Swiss PTT or Switzerland in several standardization working groups, on the topics of electronic data interchange (EDI) and message handling systems (X.400 MHS), and on the Reference Model for Open Distributed Processing (RM-ODP). His interests include software engineering and distributed systems, object modelling, systems management, CORBA, fault-tolerance and security.