Telecommunications
Information
Networking
Architecture
Consortium

TINA-C Deliverable

Issue Status: Publicly Released

Information Modelling Concepts

Version: 2.0

Date of Issue: 3 April 1995

Table 1:

Page	Company	Company Contact (Address, Telephone, Fax)

Telecommunications
Information
Networking
Architecture
Consortium

Information Modelling Concepts

DRAFT

Abstract: This document defines a set of concepts to be used in information modelling within TINA that is in line with the spirit of ODP. It also discusses the purpose and scope of information modelling and explains how an information specification is related to the specifications of the other TINA viewpoints, in particular, the computational viewpoint. The document proposes enhancements to ODP in parts where ODP is not detailed or comprehensive enough and provides guidelines based on different approaches to information modelling.

Keywords: Information modelling, information model, information viewpoint, information specification, ODP

Author(s):	**H. Christensen, E. Colban**
Editor:	**E. Colban**
Type:	**TINA Baseline**
Document Label:	**TB_EAC.001_1.2_94**
Date:	**3 April 1995**

Table of Contents

List of Figures

1. Introduction

1.1 Purpose of this Document

The TINA logical framework architecture is inspired by and based on the Open Distributed Processing Reference Model (RM-ODP, see [8]-[13]). As a result, the five ODP viewpoints (Enterprise, Information, Computational, Engineering, and Technology) have been applied. The purpose of this document is to provide a language for information modelling including a set of concepts to be used, and to define its scope and role within the TINA-C framework architecture.

1.2 Requirements for this Document

This document has to provide a set of concepts of object information modelling, such as object, object type, object class, inheritance, and relationship. It should also provide guidelines and techniques for using these concepts.

The Computing Architecture is organized along with the ODP viewpoints. ODP defines the enterprise, information, computational, engineering and technology viewpoints. The primary input to this document is therefore theRM-ODP. The importance of OSI Structure of Management Information (OSI-SMI, see [18]-[20]) has to be recognized, in order to allow for the use of large information specifications that have already been produced.

1.3 Audience

This document is primarily aimed at all persons within the TINA-C Core Team, or participating in auxiliary projects, that are involved in writing specifications of services, management applications or the DPE. The descriptions of the DPE, Services and Resource management should all comply to these principles. Anyone interested in the information modelling in TINA should also read this document.

1.4 How to Read this Document

The main body of this document defines the scope of information modelling, provides a set of concepts to be used, gives guidelines for writing information specifications, and goes through an example. It does not prescribe any concrete notation, but prescribes instead an abstract syntax that notations must conform to. Section 6 provides guidelines for using GDMO and GRM in combination as a notation, whereas Section 7 gives an example that follows these guidelines. Appendix A provides a discussion of various documents that have been used as input to this document. This discussion has been used as a basis for the results in the main body of this document. An evaluation of notations has been carried out and this is presented in Appendix B.The evaluation concludes that GDMO and GRM used in combination are a possible notation for writing information specifications, but that additional guidelines must be followed.The evaluation also concluded that Object Z is another possible notation, and an example of the use of Object-Z is given in Appendix C.

1.5 Main Inputs

The main inputs to this document are:

- Information Modelling in ODP, see [8]-[13]

- OSI Management Information Modelling, see [18],[19]

- OSI General Relationship Model, [20]

- Bellcore's Information Modeling, see [1]-[4]

- Rumbaugh et al: “Object-Oriented Modelling and Design”, see [6]

The approaches presented in these inputs are compared and discussed in Appendix A. The concepts that are presented in the main body of this document are based on that discussion. The reader is referred to Appendix A for a rationale of the choices that have been made.

1.6 Document History

This document is an update of the 1993 version of the Information Modelling Concepts. The main changes occur in Section 6 and Section 7.

Section 6 contains now a definition of the so-called Quasi-GDMO+GRM notation. The templates should now be consistent with the specifications that are generated by the Core Team with the StP tool, [28]. The justifications for the deviations from standard GDMO and GRM have been removed from this section.

2. The Scope of the Information Specifications[1]

2.1 Background

The five viewpoints or projections on distributed processing were first introduced by ANSA [6] as a result of a study on *“current practice and research in distributed computing and system design techniques... The purpose of the information projection of ANSA is the identification and location of information, and the description of information processing activities”....“Who knows what?”, and “Where can information flow”...”*([7], Volume A, Part II, Chapter 1)

The five ANSA projections have been adopted by ODP. According to ODP *“an information specification defines*

a. the informative qualities of an ODP system, i.e. the meaning that a human would ascribe to the data stored or exchanged between components of an ODP system.

b. requirements for information handling in an ODP system.”

It is unclear what exactly is meant by *information* in the above statements. Is it simply the meaning of specific data identified in the implementation of a distributed system or is it a more general knowledge of the system itself? The two seem to be inseparable, because the meaning of the data cannot be described without resorting to more general knowledge of the system (and maybe also of the environment).

Which of the following interpretations should be applied to “information” within RM-ODP?

1. the information that is communicated between the system and its users
2. the information that is communicated between the different parts or components of the system
3. knowledge about the system itself

The difference between the two first interpretations is essentially a difference of granularity or level of detail. What is seen as a system in the first interpretation can be viewed as a part of a system in a wider context; the user may be a person or some other system or part of a system. In both cases, the information specification must identify the parts, the information that each part can communicate to other parts, the flow of information between the different parts, the structure of the information, etc.

How a system is composed of smaller parts, leads to information of the system itself, i.e., the third interpretation above.

In addition to the questions that are related to the meaning of *information* as used by ANSA/ODP, are some questions about where to draw the line between the information viewpoint and the other viewpoints, in particular, the computational viewpoint.

1. We use the term *information modelling concepts* to refer to the set of concepts used in information specifications in general, whereas an information specification is often related to a specific system.

The correspondences between the information and computational specifications, their differences in concerns and their relationship has lead to much confusion[2]. One of the reasons for this is that people used to working with different paradigms keep trying to fit the “ANSA/ODP paradigm” into their own way of thinking. Questions like the following are frequently asked:

1. Does the information viewpoint correspond to analysis and the computational viewpoint correspond to design? (The analysis/design approach to system development)
2. Is the information specification *refined* into a computational specification? (The (functional) refinement approach)
3. Are information objects MOs and computational objects agents and manager objects? (OSI Network Management)
4. Is an information specification the specification of (passive) data structures whereas a computational specification is a specification of the (active) procedures? (Classical data/procedures separation approach)
5. How well does the information-computational separation actually fit with the American, Scandinavian or other object-oriented paradigms? (Object orientation paradigm)

To require that the ANSA/ODP model fit with every existing paradigm in order to justify its existence would be unreasonable. Rather than explaining the scope of each viewpoint by relating the viewpoint to the phases, stages, or whatsoever of some other “paradigm”, they should be explained within the ANSA/ODP model itself. The problem is that the existing explanations do not seem to be sufficient.

One should be careful and not undervalue the ANSA/ODP information viewpoints, without understanding its justification. Unfortunately this justification is not very well documented. The main purpose of the viewpoints is “to deal with the complexity of an ODP system,... A viewpoint leads to a representation of the system with emphasis on a specific concern... Different viewpoints address different concerns” (RM-ODP, Part 1). By this division of concerns ODP does not distinguish itself from most other systems development approaches, other than by the choice of the specific five viewpoints.

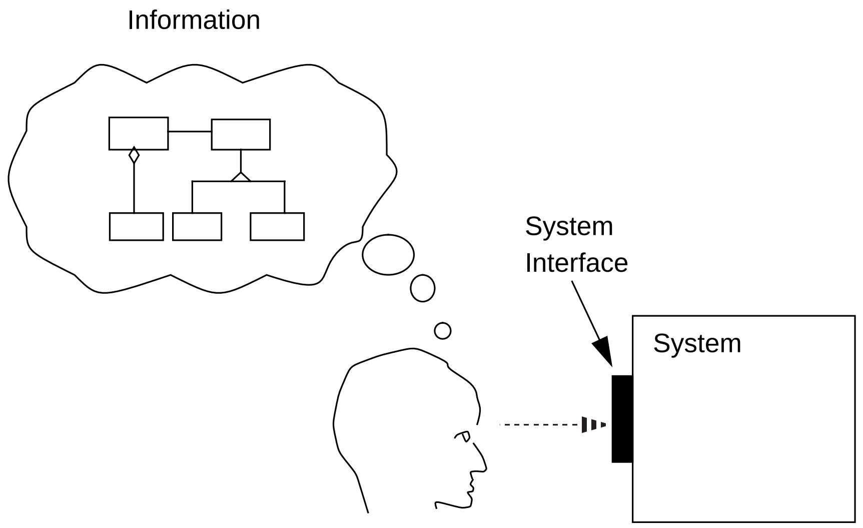
2.2 Information Specifications within TINA-C

The previous section prompts two major questions about the information viewpoint in ODP. The first one is about the meaning that should be ascribed to information, the second was about the relationship between the information viewpoint and the other viewpoints. In this section we will attempt to provide answers that are applicable to TINA. (They may not be valid for ODP although we strive to be consistent with ODP.)

2. This has at least been experienced within the TINA-C Core Team.

2.2.1 The meaning of information

When using the word *"information"* in the context of information modelling, we understand it to mean *the knowledge necessary to make appropriate use of a system or any part of the system*. In order to make appropriate use of a system, it is not necessary to understand how the system is implemented, what kind of technology is used, how the information is represented by data structures, and so on. These aspects are not dealt with in the information specification.



By information we mean the knowledge necessary to make appropriate use of a system or part of the system

Figure 2-1. Information in a TINA information specification.

Definition (of an information specification): An **information specification** is a description of a structure that models the information in a system (domain of discourse, or problem domain) in terms of information bearing entities, relationships between the entities, and constraints and rules that govern their behavior, including creation and deletion. [End of Definition]

The aim of the information specification is to identify the entities that populate the described structure, describe them, and the relationships that hold between them. The nature of the entities may vary in degree of abstraction among specifications. They are not necessarily related to any machinery that provides the mechanisms to support their behavior (we understand behavior to be the changes in the specified structure).

An area where information modelling has been fairly well developed is within network management. An agent is part of a management system used to monitor and perform management operations on a network. The agent is often implemented by a quite complex piece of software that is produced for a specific network technology. However, in order to use the agent, this complexity and technology need not be understood. The agent may have a well defined interface to a manager application. This interface can be understood in terms of a so-called *Management Information Base* (MIB). The MIB provides the structure and state of the network that the agent operates on, abstracting from implementation and technological details.

Another example of information modelling can be found in database applications. The implementation of a database may use different techniques for storing and retrieving the information, such as hash tables, arrays, keys, B-trees, advanced algorithms providing efficient search and update, etc. The information specification abstracts the information handled by the database from the techniques used to implement the database.

Every system should have a well-defined interface. A system can be decomposed into subsystems, and subsystems can be decomposed into subsystems, and so on. Each subsystem should offer a well-defined interface to the other parts of the system. A well-defined interface means that each interaction that can occur at the interface is defined and well understood. Information specifications provide the basis for *understanding* the interactions.

2.2.2 The relationship between the information and other viewpoints

Whereas the information specifications provide the necessary knowledge to interact appropriately with a system or subsystem, the computational specifications specify the system itself. This distinction draws the line between the scope of the two kinds of specifications. This line may become somewhat blurred when the structure of the information specification is reflected in the structure of the computational specification, which can be seen as a principle of good systems design. In the case where object orientation is applied, objects identified in the information specification may correspond to objects in the computational specification. For instance, a computational object may represent a piece of information and the interfaces of the computational object provide operations to act on that information. It is therefore not always possible to look at one object in isolation and tell whether it is an information or computational object. *An information object is one that appears in an information specification*, and likewise for computational objects.

Note that, although it may be a good principle, it may be difficult or even impossible to have a one-to-one correspondence between the information and computational objects without making the information specification unnecessarily complicated. Auxiliary objects may occur in the computational specification that do not correspond to any object in the information specification, but that are necessary to maintain identified relationships between the information objects. Information objects may also correspond to several computational objects. Examples are provided in [14]. There is no reason for imposing any requirement for a one-to-one mapping, nor any many-to-one mapping between information and computational objects.

Information specifications can be produced early in the analysis phase of system development since they define the information necessary for using the system and thereby also the context in which the system is going to be used. In fact, many of the information modelling concepts are borrowed from Object-oriented Modeling and Design [6], which uses the concepts in system analysis. It is however a matter of methodology, when and how the information specifications are produced. The current document does not prescribe whatsoever concerning methodology.

Information specifications may be constructed even for manual procedures, where no computational specification exist. However, there is an intimate relationship between the information modelling concepts and the **computational modelling concepts** [14]. The computational modelling concepts provide the framework for **computational specifications** of distributed systems. The abstraction level that applies to a computational specification is tightly related to the level of functionality provided by the underlying distributed processing environment. This is reflected by a richer and more specific set of concepts described in “Computation Modelling Specifications” ([14]) to deal with constraints that cannot be left out in the computational specifications. Examples of such constraints are functional separations, uniform interface descriptions, different kinds of interactions (blocked or non-blocked interrogations, announcements, transactions) etc.

Contrary to computational specifications, the abstraction level that applies to an information specification may vary from the very abstract where distribution aspects are not considered (e.g. the network is seen as one entity), to the very detailed, where some mechanisms provided by the DPE (Distributed Processing Environment) are specified (e.g. the trader).

From a methodological perspective, the information specification and the computational specification of a system can be related in one of two ways. In one, the information specification occurs earlier in the design phase, and the information specification forms the basis for the computational specification that occurs later in the design phase. This approach seems natural in data intensive applications. In the other, the information specification and the computational specification are developed together, and the information specification is used for a precise characterization of the behavior of the computational entities.

Below follow some characteristics of the information and computational specifications:

1. There is not necessarily any one-to-one mapping between information objects and computational objects. In some cases though, a computational object may represent an information object by providing interfaces for retrieving or modifying the information that is reflected by that information object.
2. There is no universal correspondence between terms (= concepts) of the two languages. For instance, the relationship concept, which is a central information modelling concept has no equivalent among the computational modelling concepts.
3. Computational objects interact (computationally) with computational objects only. This means that a computational object, e.g., does not update an attribute belonging to an information object through some computational interaction with the information object. What can happen, is that some interaction occurs between computational objects that can be explained by an update of an attribute of an information object.
4. Computational objects do not “act on” information objects, although information objects may correspond to data identified in the computational specification.
5. Interactions between the information objects are not necessarily executable. An operation of a computational object can be invoked by other objects, thereby requesting the execution of that operation. Interactions in the information

specification are most commonly modelled by relationships. A relationship may, for example, state that the creation of an information object is dependent on the existence of another object. The computational specification has to realize that dependency.

6. Any correspondence between the information specification and the computational specification must be specified in each case, so that consistency between the specifications can be ascertained. The general way of specifying the correspondence is to refer to the information specification in the documentation of the computational interfaces. In some cases, the computational specification is a refinement of the information specification.

3. Specification of Entities

3.1 Introduction

The concerns of information modelling can be divided into parts. The first consists of specifying the entities. For this part, RM-ODP, Part 2, [9], provides a fairly complete set of concepts, if the information bearing entities referred to in the above definition are modelled as objects. The second part consists of specifying the relationships between the entities and the rules that govern behavior that cannot be specified when looking at each entity in isolation from the other. For the second part one needs other concepts in additions to those provided by ODP[1]. This section aims at showing how the basic concepts of ODP can be applied to specify the entities and their behavior in isolation from other entities.

The purpose of this chapter is to describe the needs the basic concepts cover in information modelling and how they should be used in order to cover those needs. The concepts must support describing information that is contained in the various entities of the system and how this information is distributed between the entities.

Although some of the concepts are common to all viewpoints, this section defines their purpose in information modelling.

3.2 Information Objects

An **information object** is an object that occurs in an information specification. Information objects model the basic information entities in an information specification. Henceforth, whenever we write “object” without any qualifier, we mean “information object”. Each object has an identity, which is an intrinsic, invariable part of the object. Although two objects are otherwise equal, they are considered as separate objects if they have different identities.

In a model, things may happen involving one or more objects. These happenings are referred to as **actions**, or **interactions** when they involve more than one object. An object may have a state[2] that may vary over time as a result of the actions in which the object participates. The set of possible actions in which the object may participate in at a given moment in time depends on the state of the object at that moment of time.

3.3 Information Object Types

An **object type** is a predicate characterizing an object. Objects that satisfy the predicate are said to be of the given object type.[3]

1. Some clauses of ODP deal with collective behavior. Types in ODP are not restricted to be unary, which allows for a uniform approach when specifying objects and relationships.
2. The state of an object is a complex structure defined by the actual values of the variables of the object. The state space is the set of possible values the variables can take.
3. In ODP, the term *type* is used to cover predicates in a general sense, unary, binary or n-ary, “ranging over” objects, actions, and interfaces. In TINA type is used with a qualifier,e.g., object type, interface type, relationship type, etc. When the qualifier is omitted we usually mean object type.

A fundamental need covered by object orientation is abstraction. By abstraction we mean the construction of concepts by grouping entities with common characteristics and removing some of the details. For instance, the concept *person*, is formed by collecting instances of persons disregarding some details such as eye color, sex, etc.

Whereas an object is intended to model an entity, object types model concepts. The object type is specified by providing the predicate that distinguishes between instances and non instances of the type. (The predicate that corresponds to the concept of person is *"is a person".*)

In TINA, types are distinguished from **classes**, a class being the extension of a type, i.e., the set of all created, but not yet deleted, objects of a type. Since objects may be created and deleted, a class, contrary to a type, may vary over time.

In TINA, any predicate can define a type, and types that refer to the state of an object may therefore exist. As a consequence of changing its state, an object may also change its type.(For instance, a child may become an adult). The notion of type therefore does not correspond exactly to the notion of *class* as defined in most object-oriented programming languages. A class, as defined in most programming languages, can be viewed in two different ways. The first is a piece of code used to instantiate and interpret objects. This view corresponds to the TINA concept of **template**. A template is a specification of the common features of a collection of objects in sufficient detail that an object can be instantiated from it. The other view is the predicate that this piece of code expresses, i.e., the properties common to all objects that are created from that same piece of code. This is what TINA denotes a **template type**. More precisely, a template type is a type of the form "is instantiated from template *t*", where *t* is a template[4].

In an object template, an action is specified by a name, a set of input parameters and a set of return parameters. Note that an action in an information specification does not represent a means to achieve a change of state. Such "invokable operations" belong to the computational specifications. Actions in an object template specify possible state changes[5]. In addition to specifying a possible state change, an action specification also defines the "external view" of an object. The state of an object is *encapsulated* which means that it can only change through a specified action and the current state can only be deduced from the values of the parameters returned by an action.

An object template specifies when an action *can* occur and when it *must* occur. A **pre-condition** of an action is a proposition referring to the object's state that must be true in order for that action to occur. A **triggering condition** of an action is a proposition referring to the object's state such that that when it becomes true the action takes place. The state of the object after an action takes place is specified by the **post-conditions** of the action.

4. In OSI Management, the term managed object class, although defined to in a way that is consistent with the ODP definition of the term, is sometimes also used to refer to the specification of the class or to the characteristics common to a set of objects, thus not distinguishing between class, type and template.

5. Specifications of actions correspond to "dynamic schema" in ODP.

Not all actions need to be specified with triggering conditions. Actions with no triggering conditions are called **operations**. An action with triggering conditions is called a **notification** if it has at least one return parameter or an **internal action** if it has no return parameters.

By using types, objects may be classified according to common characteristics, and thereby types cover classification needs. Use of types also supports reuse, since only types need to be specified and not every single object of the type. Types also support abstraction, because objects of a given type are, in a sense, equal, disregarding specific details (identity being one of them).

3.4 Subtypes

If two types T1 and T2 are such that any (potential) object of type T2 is also of type T1, then T2 is a **subtype** of T1, and T1 is a **supertype** of T2. The super-/subtype relationship between types corresponds to implication between the predicates. The super-/subtype relationships help in organizing the classification of objects.

Whether there is a subtype relationship between two types depends solely on the types. However, in information modelling, one might want to refer to certain types (by name) and explicitly state there is a subtype relationship between them. There must then be consistency between such a statement and the specification of the types.

Subtyping should not be confused with inheritance which is a specification technique that defines a template from another template by adding or overwriting properties (attributes, behavior) (see also next section).

3.5 Inheritance

Inheritance, subtyping and specialization are concepts that are often confused or their names misused. This section tries to clarify their relationships.

The reason for considering this is that subtyping is a specification concept used to model the real world, while inheritance is a code- or specification-reuse technique. Thus subtyping is often used as the ideal for specification while inheritance represents the more pragmatic side. However, there is (or should be) a connection between the uses of the two concepts, as described below.

In Figure 3-1, an arrow indicates a relationship and the head of the arrow indicates which direction the verb of the arrow should be read. For instance: Objects Model Phenomena.

As earlier, we use ODP terminology, e.g. a type is a predicate. Figure 3.1 tries to sum up the situation. The figure states the following:

- Concepts are abstractions of phenomena. Concepts are collections of phenomena.
- Objects are instances of templates.
- Objects model phenomena, just as types model concepts.

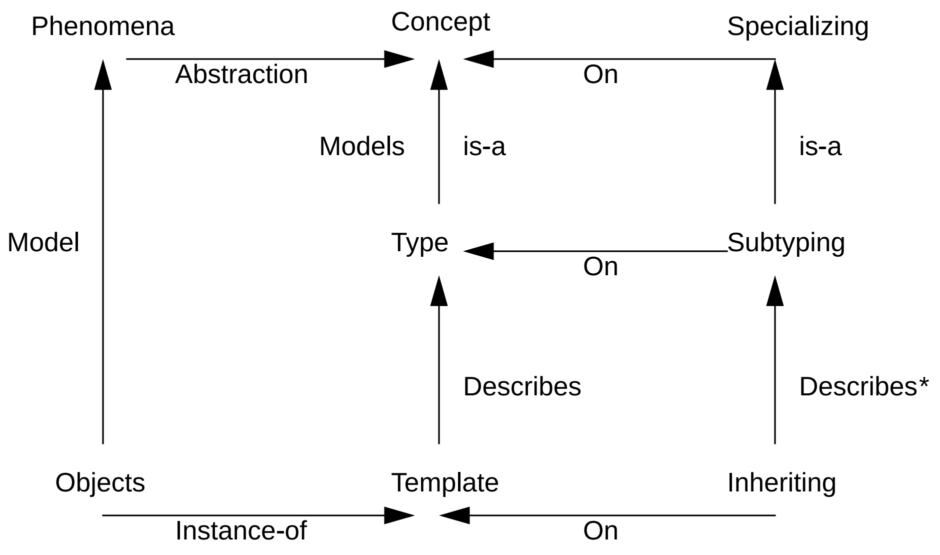


Figure 3-1. Relationships between some concepts related to inheritance

- A type is-a concept. To be exact: for every type there exist a concept which the type has an is-a relationship to. The is-a relationship means adding, and not removing properties.
- A template type describes a type.
- Specialization works on concepts; it means adding and not removing properties.
- Subtyping works on types and means implication of predicates.
- Inheritance works on template types and means code or specification re-use
- Subtyping is-a specialization; i.e. subtyping is a kind of specialization between types.
- Inheritance describes* subtyping. The * here indicates that this should be the intention: Inheritance should model subtyping to preserve polymorphism and compatibility. (This is not always the case in the actual world, however, ideally, it should be so, we claim).

The relationship between types and concepts is an is-a relationship. This means that a type is a concept. The difference is that types are predicates and therefore precisely defined, while concepts often have fuzzy borders (e.g. the concept of a chair). As an exercise try defining a chair type predicate that precisely defines the properties that make a thing a chair. Types are all *aristotelian* concepts, i.e. with precisely defined borders. Other concepts are *fuzzy* or *prototype* concepts.

Note that, in our view, inheritance should ideally model or describe subtyping. Thus, inheritance should strive to preserve all properties of the parent-type-template. In the real world, this is not always so. Often templates are inherited just for the sake of reusing specifications, not to preserve properties; this sacrifices polymorphism and behavior compatibility.

3.6 (Information Object) Templates

An object template is an expression for a template type. Object templates are used for instantiating new objects. The template used for instantiating a new object describes the properties of that object. Somehow objects and types need to be specified or described; object templates specify template types.

Since a template is an expression, it must be given in a certain language that has a certain grammar. ODP does not prescribe the use of any particular language. Some language(s) has(have) to be chosen.

The semantics of the language used for templates defines the satisfaction relation that tells when an object satisfies the specification. This satisfaction relation needs to be spelled out.

Finally, on the basis of the templates one needs to determine properties of the specified objects/types. One therefore needs a proof theory, i.e., a set of rules or calculus to decide, e.g., whether two templates specify sub-/supertypes of each other.

ODP ([11], Section 8.3) states that an object is characterized by its behavior, or dually by its state. This means that there are basically two approaches to describing an object. The first one, is to consider the object as a “black box” and only describe the set of activities (an activity is defined as a “single-headed directed acyclic graph of actions”) that the object can participate in. There is no reference to the object’s state in such a description. The other approach consists of describing the object in terms of actions and state. Each state allows the object to participate in a set of actions, each action brings the object into a new state. Indirectly this will define the set of activities that the object may participate in. We will focus on the latter approach in information specifications in TINA.

An abstract grammar for object templates is given in Section 5.3 on page 5-3.

3.7 Instantiation

Instantiation is the process which, using an object template results in a new object. An instance of a template is an object that satisfies the template, or equivalently, any object of the type that the template denotes.

An instance of a template contains information in accordance to the template. The specific information content is determined by the actual state of the object.

4. Modelling Relationships between Entities

4.1 Why Do We Need Additional Concepts

In information modelling one needs to interrelate information contained in various objects and not only the information of one object at a time. For example, suppose that certain objects represent established connections, others the available resources. If a new connection is established, the available resources are no longer the same. There is a kind of dependency between the two types of objects, which needs to be specified.

In general, the reasons for specifying relationships are:

1. Objects in a model are *not* isolated.
2. The pre- and postconditions of certain interactions often refer to more than one object.
3. Invariants may refer to more than one object.

The sub-supertype relationship in ODP is an example of a relationship that interrelates types. Other relationships are needed in addition. In the following chapters, different approaches to interrelate objects are briefly presented.

In the following sections, the terminology has extensively been borrowed from GRM. Section A.3.1 in Appendix A presents some of the major differences.

4.2 Relationships

A **relationship** is a tuple of objects related by some property that pertains to all objects of the tuple. The relationship is defined by some predicates or invariants that do not only pertain to one or the other of the objects but to the collection of the objects. The invariant may define some dependency between the objects in the relationship, e.g., the deletion of one object may imply or prevent the deletion of other objects in the relationship. There may be an invariant that interrelates attributes of different objects in the relationship. Interactions may occur between the objects in the relationship or the objects may in some manner “fit” together.

Often relationships can be deduced from the specification of each individual object in the relationship. Specifying relationships is a way of making the relationship explicit and thereby making the specification easier to understand.

The number of objects that participate in a relationship determine the “*arity”* of the relationship. If the arity is 2, the relationship is between 2 objects, and the relationship is said to be a binary relationship. Most common are binary relationships.

Just like objects, relationships may have a state. For the same reasons as for objects, relationships may have attributes, and actions may change the state.

4.3 Relationship Types

There is an equivalent relationship between relationships and **relationship types** as between objects and object types. The relationship type is a predicate over relationships used to model a concept. From the standpoint of ODP, there is no need to distinguish between objects and relationships; relationships are objects, although of a different flavor.

All instances of a relationship type are of the same arity. A position in a relationship is called a **role**. A relationship type associates each role with an object or relationship type, which means that in every instance of the relationship type each role is occupied by an object that belongs to the associated type. Note that in a relationship each role is occupied by exactly one object.[1]

Example: *Marriage* is a binary relationship type with the two roles *husband* and *wife*. The *husband* role is associated with the type *man*, *wife* is associated with the type *woman*. Every married couple is an instance of this relationship type, for instance (John, Mary). John has the role *husband*, and that is OK, because *husband* is associated with *man* and John is a *man*.

Note that a relationship may be bound to a role in another relationship and thus be regarded as an object. For instance a subscription is a relationship between a user and a service, while a subscription may be an object in another relationship, e.g. billing. So, both relationships and objects are first class objects (of the same kind).

Of particular interest in a relationship type are constraints on the creation and deletion of relationship instances and the participating objects. In the above example, the words "until death do us part" express that the relationship cannot be deleted before either John or Mary are "deleted".

4.4 Role Cardinality

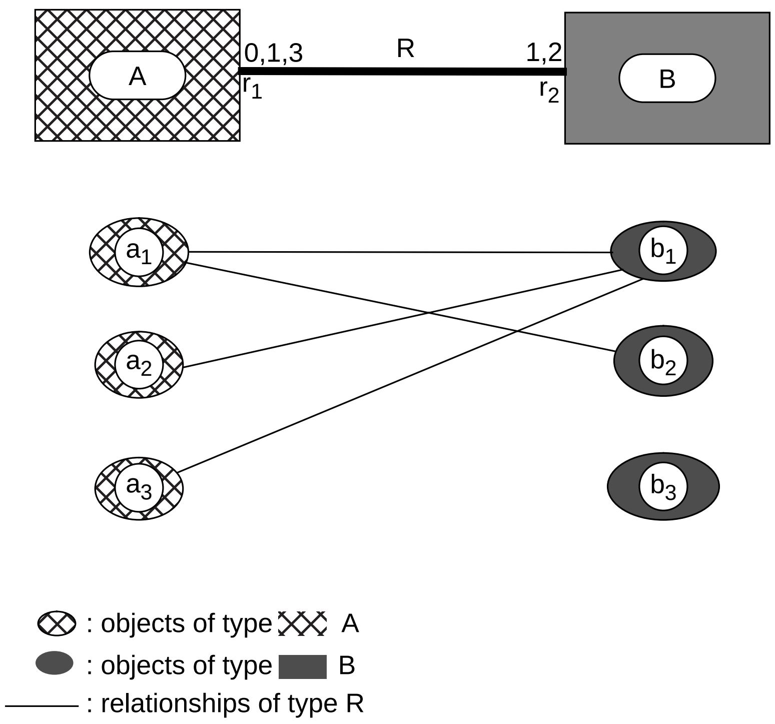
In addition to specifying constraints on the (individual) instances of a relationship one often needs to specify constraints on the *class* of all instances of a relationship. Such a constraint could, e.g., restrict the number of instances of a type. In general, one should be cautious when specifying such constraints, especially in the context of large distributed systems, due to the general lack of control that is needed to meet such universal constraints.

Given a binary relationship type and a role, the **role cardinality** is a set of non-negative integers associated with the role which constrains the number of relationships of the given relationship type that have the same object in the other role. More precisely, suppose R is a relationship type with roles r_1 and r_2 with associated types T_1 and T_2, respectively, and S_1 is a set of non-negative integers specifying the role cardinality of r_1. The role cardinality constraint is met if for every instance o_2 of type T_2 the number of relationships in which o_2 is in role r_2 is some member of S_1. Note that this may be counter-intuitive since the role cardinality of r_1 is not a constraint on the number of relationships a T_1 object participates in (see example below).[2]

1. In GRM a set of objects may be bound to a role.

Example 1: If polygamy is banned, the role cardinality of both roles *husband* and *wife* of the relationship type *marriage* is {0,1}.The role cardinality of *husband* is {0,1}, because every woman can be wife in zero or one marriage relationship. In other words, the role cardinality of husband is {0,1}, because every woman can be related to zero or one husband. Likewise for the role cardinality of wife.

Example 2: The figure shows an example with 6 objects.



The binary relationship type relates types A and B. The role r_1 is associated with type A and the role r_2 is associated with type B. The actual set of relationships is the set {(a_1,b_1), (a_1,b_2), (a_2,b_1), (a_3,b_1)}. This makes the following role cardinalities valid: S_1={0,1,3} and S_2={1,2}. Both role cardinality sets S_1 and S_2 can be extended and the role cardinalities will still be valid.

Figure 4-1. Example of a binary relationship type.

4.5 Generic Relationship Types

A **generic relationship type** is a relationship type where the roles are not associated with types. Such generic relationship types are bound into specific relationship type by associating the role to types. Binding generic relationship types into specific relationship types is referred to as **role binding**.

2. In GRM, role cardinality is defined as the number of objects that are bound to a role

A typical example of a generic relationship type is *composition*. Books are composed of chapters, trains are composed of cars. These are two examples of relationships types obtained by associating the *composite* role to the type *book* and the role *component* to the type *chapter* in the first case, and the role *composite* to the type *train* and the role *component* to the type *car* in the second case.

Experience has shown that a relatively small number of such generic relationship types can be used to instantiate a large number of relationship types.

4.6 Relationship Attributes

Just as objects may have attributes, so can relationships. Such attributes contain information that do not naturally belong to any of the participating objects, but to the relationship itself. For instance, the wedding date is an attribute of the *marriage* relationship.

4.7 Relationship Actions

Usually, an interaction will be specified in the object templates of one of the participating objects. However, such an interaction may not fit naturally as part of any of the participating objects. For instance, a document can be printed on a printer and a *print* operation can be specified either in the object template describing the *document* type or in the template describing the *printer* type (or both). A third possibility is to specify a relationship type *printable_on* with roles *printee* and *printer* associated with types *document* and *printer* respectively, and specify *print* as an operation of that relationship type. Operations, notifications and internal actions that are specified is such a manner are relationship actions. Another example is a Replica, which is a relationship between an object and a replicated object. Changing the degree of consistency is a relationship action. There may be paradigmatic or style constraints that prohibit relationship actions, but this document does not provide any recommendations in either direction regarding the specification of relationship actions.

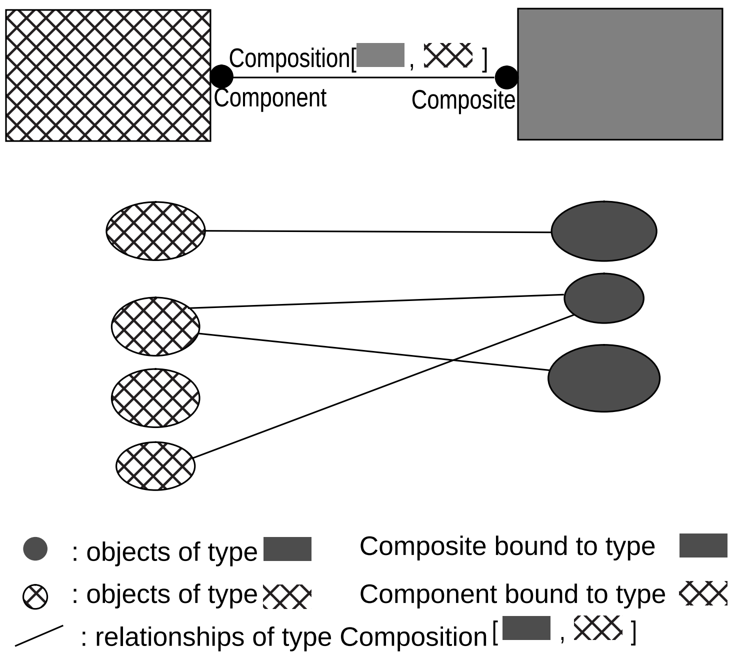
4.8 Relationship Templates

A relationship template is a description of a relationship type. The relationship between a relationship template and a relationship is exactly the same as between an object template and an object, see Section 3.6 on page 3-5. Section 5 provides an abstract grammar for relationship templates, as well as abstract grammars for generic relationship templates and role binding templates.

4.9 Composition and Containment

Many relationship types that occur often in information specifications can be derived from a few generic relationship types. Composition and Containment are two such generic relationship types.

4.9.1 Composition



The top part of the figure illustrates a role binding of the Composition relationship type. The ovals are objects, the lines connecting the oval are relationships.

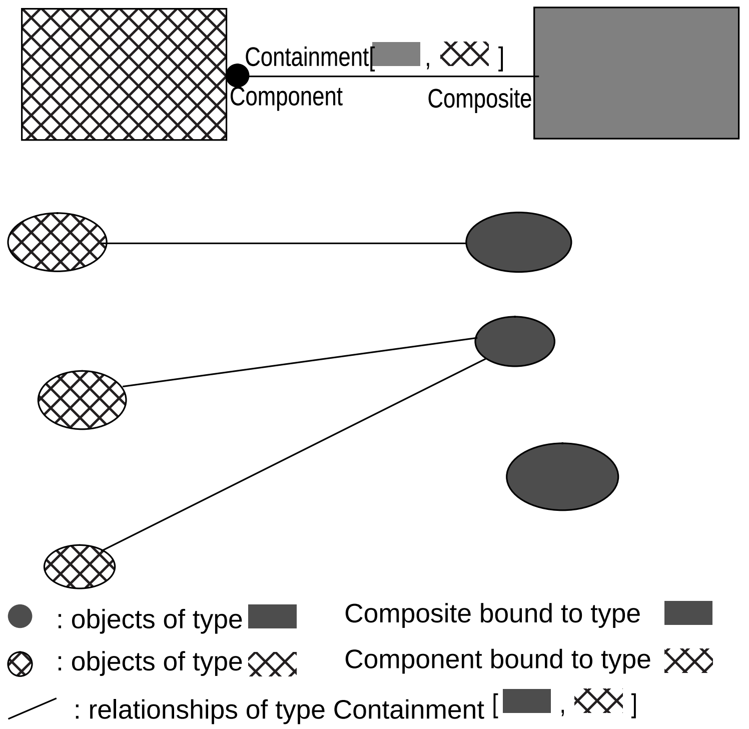
Figure 4-2. An example of the Composition relationship.

Composition is a generic relationship type with no constraints other than that it must be binary and irreflexive (i.e., the same object cannot be bound to both the Component and Composite roles in one instance of the relationship type). It has two roles; *Composite* and *Component*. In order to use this generic relationship type, the roles must be associated to specific types by a role binding. Usually, instances of the resulting relationship type exist between objects where one (the one bound to the Component role) can be viewed as “part of” the other (the one bound to the Composite role). Instances of the type bound to the Component role may or may not exist independently of any composition. This may be further restricted in the role binding. Likewise, a component may be part of multiple composites. An object can have components of different types, in which case one will use multiple role bindings.

Often a component (i.e., an object that is bound to the Component role) may be the Composite in another composition. Composition may be recursive which means that the same type is bound to both roles.

The component object and the composite object in a composition may share some properties. These properties can be specified either in the role binding or by specifying generic relationship subtypes of Composition.

4.9.2 Containment


Containment[▇ , ▨]
Component
Composite
: objects of type ▇
: objects of type ▨
Composite bound to type ▇
Component bound to type ▨
: relationships of type Containment [▇ , ▨]

The top part of the figure illustrates a role binding of the Containment relationship type. The ovals are objects, the lines connecting the oval are relationships.

Figure 4-3. An example of the Containment relationship

Containment[3] is a generic relationship type which is a subtype of Composition. The only thing that distinguishes it from Composition is the role cardinality of the Composite role which is {1}. This means that if a type T_1 is bound to the Component role and T_2 is bound to the Composite role in a role binding, every instance of T_1 is contained in exactly one instance of T_2.

3. Containment in OSI is not a generic relationship type, but a principle saying that every managed object (except a root object) is contained in exactly one other managed object. The name binding identifies possible superior and subordinate classes. Different objects of the same class may use different name bindings. A possible use of the generic relationship types Composition and Containment in the context of OSI network management is to bind the superior class (type) to the Composite role and the subordinate class (type) to the Component role. Whenever every instance of the subordinate class is contained (in the sense of OSI) to an instance of the same superior class (i.e., the subordinate class is in only one name binding), the Containment relationship type is used, otherwise the Composition relationship type is used.

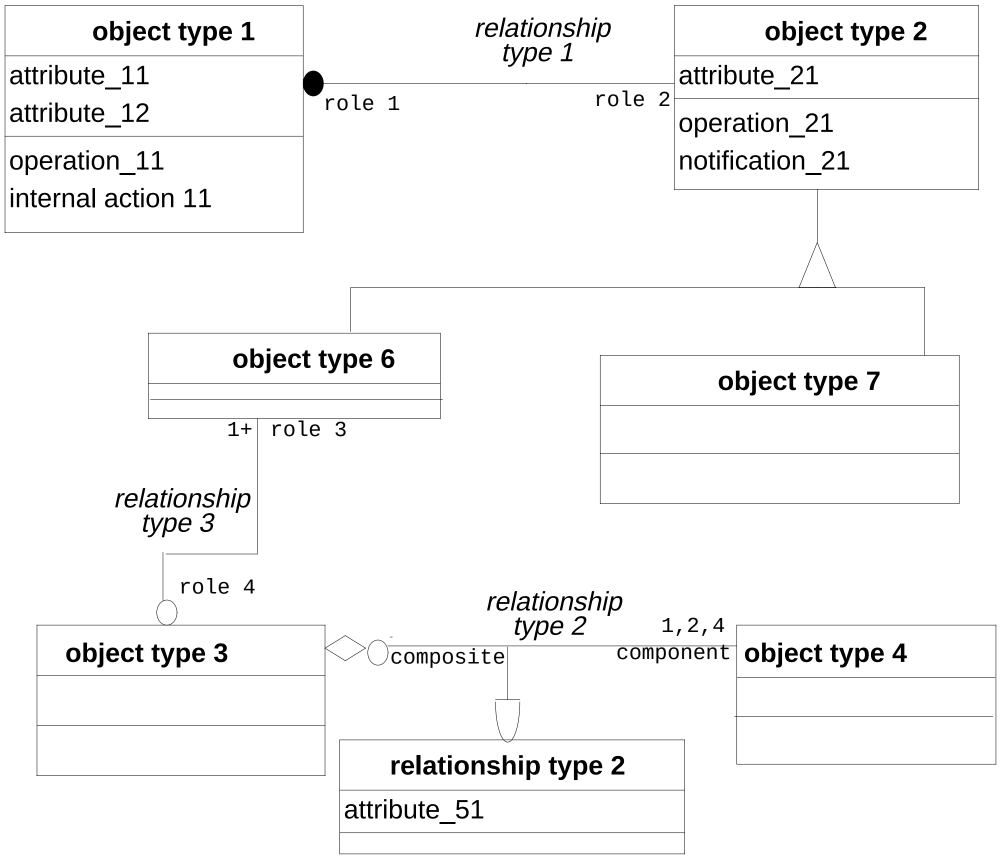
5. Prescriptions on Notation

Section 3 and Section 4 provide a set of concepts to be used in information specifications. A notation and guidelines for writing specifications have yet to be provided. This is the aim of this section.

Due to the problems that the prescription of a concrete notation would cause , an abstract grammar is given for object templates, relationship templates, generic relationship templates, and role binding templates. Quasi-GDMO+GRM is one concrete notation for these templates. Only the abstract grammar is prescribed to be used within TINA.

5.1 Identifying Object Types and Relationship Types

An information specification should at least identify the object types and relationship types that populate the specification. This can be done in a graphical manner using OMT's notation. The notation is provided in Figure 5-1, the concepts are explained in previous sections. OMT's naming of the concepts have been changed in accordance with the names provided in the previous sections.



This diagram is an example of an OMT diagram. Relationship type 1 has two roles; role 1 and role 2 associated with object types 1 and 2 respectively. The role cardinality of role 1 is {0,1,2,...}, symbolized by a filled small circle; the role cardinality of role2 is {1} which is the default. Object types 6 and 7 are subtypes of object type 2, which is symbolized by the triangle. A filled triangle would symbolize that the two subtypes could have a common instance. Role 3 has role cardinality {1,2,...} whereas role 4 has role cardinality {0,1}. Relationship type 2 is a composition relationship, which is symbolized by the diamond. A component can be related to 0 or 1 composite, which is symbolized by the open circle next to the diamond. Every composite is composed of 1, 2, or 4 components. Relationship type 2 has one relationship attribute which is attribute_51.

Figure 5-1. OMT's graphical notation.

5.2 Abstract Grammars

An abstract grammar defines a set of constructs in terms of other constructs. A construct defines the structure of a set of syntactic objects, called specimens of the construct. Each construct is associated with a production that has the form

T == right-hand-side

where T is a construct. Constructs are written with capital initials, and each construct is defined by at most one production. Constructs that are not defined are written in boldface. In the right hand side of a production, constituents are separated by “;”. We also use the Kleene *-operator to mean that zero or more specimens of the construct that it is appended to may appear in a specimen of the defined construct. The +-operator is used almost in the same way, to mean one or more specimens. There is no ordering implied for the constituents of a construct. A concrete syntax can be added to an abstract syntax by adding keywords and ordering constraints (what is commonly referred to as syntactic sugar). The reader is referred to [26] for more details.

5.3 Object Template

Object_Template	==	n : **Name**; Inherited_Template*; s : State; Operation*; Notification*; Internal_Action*
Inherited_Template	==	n : **Name**
State	==	Attribute*; Initial_Predicate*; Invariant*
Attribute	==	n : **Name**; t : **Attribute_Type**
Initial_Predicate	==	p : **Predicate**
Invariant	==	p : **Predicate**
Operation	==	s : Operation_Signature; p : Pre_Condition; q : Post_Condition
Operation_Signature	==	n : **Name**; Parameter*; Result*
Parameter	==	n : **Name**; t : **Parameter_Type**
Result	==	n : **Name**; t : **Result_Type**
Pre_Condition	==	p : **Predicate**

Post_Condition	==	*p :* ***Predicate***
Notification	==	*s : Notification_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Notification_Signature	==	*n :* ***Name****; Result+*
Triggering_Condition	==	*p :* ***Predicate***
Internal_Action	==	*s : Internal_Action_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Internal_Action_Signature	==	*n :* ***Name***

In this grammar ***Name***, ***Predicate***, ***Attribute_Type***, ***Parameter_Type***, and ***Result_Type*** are not defined. ***Names*** could be defined as a string of characters, but they could have some further structure depending on a naming convention that is applied. Naming is a topic for further study in TINA. How ***Predicates*** are structured depends on the language that is chosen to write the predicates. This language may be informal English, a first order language, or some other formal language. How attribute, parameter and result types are structured also depends on the actual notation that is chosen. They could be simple names of data or object types, or they could be specified using some type constructors such as the powerset, choice or sequence operator.

Each *Attribute* is specified by a *Name* and an *Attribute_Type*. In every instance of the template the attribute has a value of the specified type. The values of all the attributes of an object defines that object's state. The values of the attributes are not visible or subject to modification unless an operation is provided for that purpose[1].

The *Initial_Predicate*s specify the values of the attributes of an object when it is instantiated. It is not necessary to specify the initial state; one can simply assume that the attributes have values according to their types and the invariants of the object.

The *Invariants* are predicates that must always hold for every instance of the template. They refer to the values of the attributes and constrain the possible set of states of the object (the state space).

Whereas objects model entities, *Operations, Notifications* and *Internal Actions* model the way the modeled entity can change state. Operations are specified by their *Operation_Signature* and their semantics. The *Operation_Signature* consists of the operation name, the input *Parameters* and the *Results*. More than one result is possible.

1. The term *attributes* is interpreted differently in GDMO. In GDMO, an attribute defines one or more operations (GET, SET, REPLACE) to manipulate parts of the state of the managed object.

The *Pre- and Post_Conditions* are predicates that specify the semantics of the operations. They do not have to define the modeled operation deterministically; non-determinism is often a desired feature in information specifications, as opposed to computational specifications. Pre-conditions are predicates that have to hold prior to the operation, post-conditions are predicates that hold after execution of the operation. Pre- and postconditions only refer to the state of the object that the operation is associated with. If an operation has effects on more than one object, then there is a relationship between the involved objects. The side effects of an operation on another object are specified in the relationship template.

A *Notification* models an action that is triggered and that returns at least one prameter. In addition to the pre- and post-conditions there is a *Triggering_Condition*. The notification takes place when the *Triggering_Condition* becomes true. A *Pre_Condition* of a notification is a predicate that must be true when the notification is emitted. A *Triggering_Condition* of a notification is a predicate that when true, makes the object emit the notification. For instance, a clock can be modeled as an object with a notification called ALARM. A pre-condition is that the alarm is set. A triggering condition is that the alarm time equals the displayed time.

An *Internal_Action* models an action that is triggered and that has no return parameters. Internal actions can be used to specify changes in the objects state that are not caused by any operation or notification. Every change of state should be accounted for by either an operation, notification or internal action in order to preserve the encapsulation property.

An object type may be specified by using inheritance, in which case all attributes, operations, notifications and internal actions of the *Inherited_Template* become attributes, operations, notifications and internal actions of the current template. Invariants are conjoined, so are pre- and postconditions of the same operation, notification, etc. Triggering conditions are disjoined. The reason for these rules is polymorphism. We want inheritance to preserve properties of the parent type. For instance, if invariants are conjoined, all invariants will be preserved in the inherited type.

5.4 Relationship Template

Relationship types (generic or specific) can be specified using a relationship template. A specific relationship type can be specified by a relationship template in combination with a role binding template.The structure for a relationship template is:

Relationship_Template	==	n : **Name**; Inherited_Template*; Role*; Class_Constraint*; s : State; Operation*; Notification*; Internal_Action*
Inherited_Template	==	n : **Name**
Role	==	n : **Name**; t : Associated_Type; [s : Role_Cardinality]
Associated_Type	==	n : **Name**
Role_Cardinality	==	**Set_Of(Integer)**

Class_Constraint	==	*p :* ***Predicate***
State	==	*Attribute*; Initial_Predicate*; Invariant**
Attribute	==	*n :* ***Name***; *t :* ***Attribute_Type***
Initial_Predicate	==	*p :* ***Predicate***
Invariant	==	*p :* ***Predicate***
Operation	==	*s : Operation_Signature; p : Pre_Condition; q : Post_Condition*
Operation_Signature	==	*n :* ***Name***; *Parameter*; Result**
Parameter	==	*n :* ***Name***; *t :* ***Parameter_Type***
Result	==	*n :* ***Name***; *t :* ***Result_Type***
Pre_Condition	==	*p :* ***Predicate***
Post_Condition	==	*p :* ***Predicate***
Notification	==	*s : Notification_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Notification_Signature	==	*n :* ***Name***; *Result+*
Triggering_Condition	==	*p :* ***Predicate***
Internal_Action	==	*s : Internal_Action_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Internal_Action_Signature	==	*n :* ***Name***

A relationship type is specified almost in the same manner as an object type. The ***Predicates*** that occur in *Initial_Predicates, Invariants, Triggering_Conditions, Pre_ and Post_Conditions* may refer to the *Attributes* of the relationship as well as to the *Attributes* of the objects that are bound to the roles of the relationship. Their interpretation is the same as for the corresponding predicates in an object template.

The *Inherited_Template* is the name of a relationship template where each role maps onto exactly one role of the current template. The *Associated_Types* of the current template must be subtypes of the *Associated_Types* of the corresponding roles of the *Inherited_Template* if the *Inherited_Template* is not generic. The *Role_Cardinalities* are intersected, the *Class_Constraints* are conjoined; otherwise the same rules of inheritance apply as for inheritance in object templates. Again, the reason for this is polymorphism.

The *Class_Constraints* are predicates that refer to relationship template class, i.e, the set of all instances of the template or to the classes of the *Associated_Type*s of the *Roles*[2]*.* The *Class_Constraints* can express conditions for creating or deleting instances of these classes. The *Class_Constraints* may be structured into invariants, pre- and post-conditions for creating an deleting instances of the relationship (relationship constraints) template, and pre- and post-conditions for creating and deleting instances that are bound to the roles (role constraints).

A *Role* is specified by a ***Name***, an *Associated_Type* and the *Role_Cardinality*. The *Associated_Type* is specified by the name of an *Object_Template* or a *Relationship_Template.* The *Role_Cardinality*, which is optional, is a set of non-negative integers. How this set is specified, depends on the concrete syntax that is chosen. It could, e.g., be specified as an interval or by enumerating the elements of the set.

5.5 Generic Relationship Template

*Generic_Relationship_Template==n : **Name**; Inherited_Template*; Role*; Class_Constraint*; s : State; Operation*; Notification*; Internal_Action**

Inherited_Template	==	*n :* ***Name***
Role	==	*n :* ***Name****; [s : Role_Cardinality]*
Role_Cardinality	==	***Set_Of(Integer)***
Class_Constraint	==	*p :* ***Predicate***
State	==	*Attribute*; Initial_Predicate*; Invariant**
Attribute	==	*n :* ***Name****; t :* ***Attribute_Type***
Initial_Predicate	==	*p :* ***Predicate***
Invariant	==	*p :* ***Predicate***

2. *Role_Cardinality* could alternatively be specified as a *Class_Constraint*.

Operation	*==*	*s : Operation_Signature; p : Pre_Condition; q : Post_Condition*
Operation_Signature	*==*	*n :* ***Name****; Parameter*; Result**
Parameter	*==*	*n :* ***Name****; t :* ***Parameter_Type***
Result	*==*	*n :* ***Name****; t :* ***Result_Type***
Pre_Condition	*==*	*p :* ***Predicate***
Post_Condition	*==*	*p :* ***Predicate***
Notification	*==*	*s : Notification_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Notification_Signature	*==*	*n :* ***Name****; Result+*
Triggering_Condition	*==*	*p :* ***Predicate***
Internal_Action	*==*	*s : Internal_Action_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Internal_Action_Signature	*==*	*n :* ***Name***

The *Generic_Relationship_Template* is almost identical in structure to the *Relationship_Template*, except that no type is associated to the *Roles*. An *Inherited_Template* must be the name of a *Generic_Relationship_Template.*

5.6 role binding Template

Relationship_Binding_Template==n : ***Name****; t : Generic_Relationship_Template_Name; Role*; Class_Constraint*; s : State; Operation*; Notification*; Internal_Action**

Generic_Relationship_Template_Name==n : Name

Role	*==*	*n :* ***Name****; t : Associated_Type; [s : Role_Cardinality]*

Associated_Type	==	*n :* ***Name***
Role_Cardinality	==	***Set_Of(Integer)***
Class_Constraint	==	*p :* ***Predicate***
State	==	*Attribute*; Initial_Predicates*; Invariant**
Attribute	==	*n :* ***Name****; t :* ***Attribute_Type***
Initial_Predicates	==	*p :* ***Predicate***
Invariant	==	*p :* ***Predicate***
Operation	==	*s : Operation_Signature; p : Pre_Condition; q : Post_Condition*
Operation_Signature	==	*n :* ***Name****; Parameter*; Result**
Parameter	==	*n :* ***Name****; t :* ***Parameter_Type***
Result	==	*n :* ***Name****; t :* ***Result_Type***
Pre_Condition	==	*p :* ***Predicate***
Post_Condition	==	*p :* ***Predicate***
Notification	==	*s : Notification_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Notification_Signature	==	*n :* ***Name****; Result+*
Triggering_Condition	==	*p :* ***Predicate***
Internal_Action	==	*s : Internal_Action_Signature; t : Triggering_Condition; p : Pre_Condition; q : Post_Condition*
Internal_Action_Signature	==	*n :* ***Name***

The *Relationship_Binding_Template* describes a specific relationship type given a generic relationship type. The *Relationship_Binding_Template* specifies

- the name of the generic relationship type that is being bound
- the types that are associated with each role
- additional class constraints (optional)
- additional relationship attributes (optional)
- additional relationship operations, notifications, and internal actions (optional)

The ***Name*** of the *Relationship_Binding_Template* is the name of the resulting relationship type.

6. Quasi-GDMO+GRM

As mentioned in the evaluation of GDMO and GRM in Appendix B, these notations will necessitate some guidelines of use in order to obtain consistency with the concepts outlined in this document. In this chapter we provide a concrete notation for writing information specifications. We call this notation Quasi-GDMO+GRM because of its resemblance to the GDMO and GRM notations.

We stress that we are not applying concepts of OSI MIM, we are only using the notation without any implications on the TINA-C information modelling concepts. Whatever meaning is associated with the keywords in the GDMO and GRM template structure in the context of OSI MIM does not apply here. This section has to provide the new meanings of the keywords.

6.1 Object Type Specifications

This section gives guidelines for how to specify object types.

6.1.1 Object Type Template Structure

The following is the recommended object type template.

```
<object-type-label> OBJECT TYPE
  [DERIVED FROM <object-type-label> [,<object-type-label>]*;]
  CHARACTERIZED BY <package-label> PACKAGE
  BEHAVIOUR <behavior-definition-label> BEHAVIOUR DEFINED AS
      “
      [COMMENTS: <informal explanations that may ease the understanding of
      the specification>;]
      [INVARIANT: <invariant properties of the object>;]
      [object-constraints]
      [[attribute-comments][attribute-operations]]*
      [operation-spec]*
      “;
  ATTRIBUTES
      [attribute-declaration [, attribute-declaration]*];
  ACTIONS
      [action-declaration [, action-declaration]*];
  NOTIFICATIONS
      [notification-declaration [, notification-declaration]*];
REGISTERED AS ??;
```

6.1.1.1 Supporting productions

```
object-constraints  ->    [<obj-create-signature>
```

```
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
                           [<obj-delete-signature>
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
attribute-comments  ->     <attribute-name> <comments>;
attribute-operations->     {get | replace | get replace |
                           add | remove | add remove}
get                 ->     <attribute-name>-GET(): (x: attribute-type)
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
replace             ->     <attribute-name>-REPLACE(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
add                 ->     <attribute-name>-ADD(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
remove              ->     <attribute-name>-REMOVE(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
operation-spec      ->     <operation-signature>
                           [<commments>;]
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
                           [TRIGGERINGCONDS: <triggering-conditions>;]
attribute-declaration->    <attribute-name>
                           [PERMITTED VALUES: <attribute-type>]
                           [INITIAL VALUE: <initial-value>]
                           [GET | REPLACE | GET-REPLACE]
                           [ADD | REMOVE | ADD-REMOVE]
action-declaration  ->     operation-signature
notification-declaration-> operation-signature
operation-signature ->     <operation-label>
                           ([<var>: <type>[,<var>: <type>]*]):
                           ([<var>: <type>[,<var>: <type>]*])
```

6.1.1.2 Supporting Definitions

The `DERIVED FROM` clause is optional; multiple inheritance is permitted. The behavior of the derived object type is the conjunction of the behavior of all inherited types and its own specified behavior. There are no precedence rules in case of inconsistencies; inconsistencies result in invalid specifications. The set of actions and notifications of a derived type is the union of the actions and notifications of all inherited types and the actions and notifications of the derived type. Invariants of the super-types are conjoined to the invariant of the sub-type. In case an attribute, action or notification is specified more than once, the post-conditions become the conjunction of all the specified post-conditions. The triggering conditions are the disjunction of all the specified pre-conditions and triggering conditions respectively.[1]

`Object-constraints` are constraints on the creation and deletion of instances of the object template. These constraints are specified as pre- and post-conditions of the create and delete operations. The default signature of the create operation is

```
create-<object-type-label>():(x: <object-type-label>)
```

In the pre- and post-conditions `x` is used to refer to the new object instance. The default signature of the delete operation is

```
delete-<object-type-label>(x: <object-type-label>):()
```

where `x` refers to the deleted object.

`Attribute-comments` provide informal explanations of the purpose of an attribute. `Attribute-operations` specify the `GET, REPLACE, ADD, REMOVE` operations. If, e.g., an attribute `attr1` is declared as `GET` in the `attribute-declaration`, and `type1` is the type of `attr1`, then `attribute-operations` will contain the specification of the operation

```
GET-attr1():(x: type1)
```

This operation is specified by the pre- and post-conditions of the operation.

Attributes are declared under the `ATTRIBUTES` part of the template. The `GET, REPLACE, ADD, REMOVE` operations are also declared here. Attributes are declared by their name. The `PERMITTED VALUES` clause is used for specifying the type of the attribute. The `<attribute-type>` is a name that refers to the definition of the attribute type provided elsewhere.The definition of the attribute types can be given using ASN.1 The `INITIAL VALUE` clause specifies the value of the attribute when a new instance is created.

All operations other than the `GET, REPLACE`, etc., operations, are declared either as actions or notifications. The signature is given by the action or notification name, a list of input arguments and a list of return values. The type of each argument and return value is specified in the same way as the types of attributes, i.e., by a name that refers to some definition of the type. The types can be specified using ASN.1. Notifications are distinguished from (other) actions by the fact that they have return value(s), and that they are triggered, i.e., a triggering condition is specified. Declared actions and notifications are specified in the `BEHAVIOUR` part of the object type template; each `operation` is the specification of a declared action or notification.

1. Implicit pre-conditions are conditions that have to be met prior to the execution of an action in order to preserve the invariant of the object. Such implicit pre-conditions may be stronger in the sub-type than in the super-type, since the invariant of the sub-type is stronger than the invariant of the super-type. However, if we require that actions in a sub-type be *refinements* of their counterpart in the supertypes, then pre-conditions in the sub-type should be weaker than the pre-conditions in the super-types.

6.2 Relationship Type Specifications

This sections give guidelines for how to specify relationship types.

6.2.1 Relationship Type Template Structure

```
<relationship-type-label> RELATIONSHIP TYPE
  [DERIVED FROM <relationship-type-label>[, <relationship-type-label> ]*;]
  CHARACTERIZED BY <relationship-package-label> PACKAGE
  BEHAVIOUR <behavior-definition-label> BEHAVIOUR DEFINED AS
      "
      COMMENTS: <informal explanations that may ease the understanding of
      the specification>
      INVARIANT: <invariant properties of the relationship>
      [relationship-constraints]
      [role-constraints]
      [[attribute-comments][attribute-operations]]*
      [operation-spec]*
      ";
  ATTRIBUTES
      [attribute-declaration [, attribute-declaration]*];
  ACTIONS
      [action-declaration [, action-declaration]*];
  NOTIFICATIONS
      [notification-declaration [, notification-declaration]*];
  [ROLE <role-label> roleproperties;]*
REGISTERED AS ??;
```

6.2.1.1 Supporting Productions

```
relationship-constraints-> [<relationship-create-signature>
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
                           [<relationship-delete-signature>
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
role-constraints    ->     [<role-create-signature>
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
                           [<role-delete-signature>
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]]
attribute-comments  ->     <attribute-name> <comments>;
attribute-operations->     {get | replace | get replace |
                           add | remove | add remove}
get                  ->    <attribute-name>-GET(): (x: attribute-type)
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
replace              ->    <attribute-name>-REPLACE(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
add                  ->    <attribute-name>-ADD(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
remove               ->    <attribute-name>-REMOVE(x: attribute-type): ()
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
operation-spec       ->    <operation-signature>
                           [<commments>;]
                           [PRECONDS: <pre-conditions>;]
                           [POSTCONDS: <post-conditions>;]
                           [TRIGGERINGCONDS: <triggering-conditions>;]
attribute-declaration->    <attribute-name>
                           [PERMITTED VALUES: <attribute-type>]
                           [INITIAL VALUE: <initial-value>]
                           [GET | REPLACE | GET-REPLACE]
                           [ADD | REMOVE | ADD-REMOVE]
action-declaration  ->     operation-signature
notification-declaration-> operation-signature
operation-signature ->     <operation-label>
                           ([<var>: <type>[,<var>: <type>]*]):
                           ([<var>: <type>[,<var>: <type>]*])
roleproperties       ->    ROLE CARDINALITY CONSTRAINT (<lower>..<upper>)
```

6.2.1.2 Supporting Definitions

Multiple inheritance of relation type specifications is permitted using the `DERIVED FROM` clause. Same rules applies as for object type specifications.

`BEHAVIOUR` is defined as for object types. The conditions (triggering, pre-, post-) associated with the actions (including the `GET, REPLACE` operations on the `ATTRIBUTES`, if any) of the specified relationship are given in the behavior part.

The `relationship-constraints` are constraints on the creation and deletion of instances of the relationship template and are specified by pre- and post-conditions. The default signature of the relationship creation is

```
create-<relationship-type-label>(x1: <role-label>,...,xn: <role-la-
bel>):(y: <relationship-type-label>)
```

The <role-labels> in the signature can be seen as generic types to be determined in the role bindings. The default signature of the relationship deletion is

```
  delete-<relationship-type-label>(x: <relationship-type-label>):()
```

The `role-constraints` are constraints on the creation and deletion of the instance bound to each role. The default creation signature is

```
  create-<role-label>():(x: <role-label>)
```

The x in the signature refers to the object bound to that role. The default deletion signature is

```
  delete-role-label>(x: <role-label>):()
```

The `role-constraints` of a role are additional to the `object-constraints` specified in the templates of the object types bound to that role.

The `ATTRIBUTES` clause does not exist in the GRM templates, but has been added here.The values of the attributes of a relationship determine the state of the relationship (see “link attributes” in [6]).

The `ACTIONS` clause has been added as well. These are actions that are more naturally associated with the relationship as a whole rather than to any particular role or particular object participating in a role of the relationship.

A role is a generic type parameter and a name to refer to the object in a relationship instance that participates in that role. For instance, if a relationship type defines a role *r1*, and *x* is an instance of the relationship type, then *x.r1* refers to the object that play the role r1 in *x*. A role is also a generic type parameter that is bound to an object-type through a role binding.

6.2.2 Role Binding Template Structure

```
<role-binding-label> ROLE BINDING
  RELATIONSHIP TYPE <relationship-type-label>
      [ROLE <role-label>
        RELATED TYPES {<object-type-label> | <relationship-type-label>};]*
REGISTERED AS ??;
```

6.2.2.1 Supporting Definitions

The value of `RELATIONSHIP TYPE` is the label of the relationship type that the binding applies to.

A role may be bound to either an object type or a relationship type. The type a role is bound to is specified as the `RELATED TYPE` of that role.

6.3 Important Differences between GDMO/GRM and quasi-GDMO-GRM

1. quasi-GDMO-GRM is not oriented towards CMIP.
2. There is no assumption of an agent in quasi-GDMO-GRM.
3. Operations are not invokable in quasi-GDMO-GRM; operations in quasi-GDMO-GRM specify possible state changes.
4. There is no *name binding* in quasi-GDMO-GRM.
5. Different parts of the specification of object types in quasi-GDMO-GRM are kept together; packages, attributes, operations are kept within the object template.
6. There are no conditional packages in quasi-GDMO-GRM
7. In quasi-GDMO-GRM, attributes are specified by an attributes name and an attributes type. The scope of the attribute name is local to the object template.
8. In quasi-GDMO-GRM, operations and notifications are specified giving the input parameters and return parameters. The scope of the operation name is local to the object template.
9. Relationships specified in GRM are implemented in GDMO by means of relationship attributes, relationship objects, or name bindings; this is not the case in quasi-GDMO-GRM.
10. Role cardinality has a different interpretation in quasi-GDMO-GRM, see discussion in Section A.3.1.

6.4 Issues for Further Study

In Section 5, the role binding template allowed for constraints, attributes and actions that are additional to the ones given in the bound relationship type. The role binding template given in this section has been simplified not to allow for such additional specifications. Can one live with this simplification?

Naming is a general issue for further study in TINA-C. The results of such a study will affect the naming of the object and relationship types. The `object-identifiers` used in the `REGISTERED AS` clauses will be determined by the TINA naming policy.

7. Example

This example contains some of the Network Resource Information Model Specification of TINA-C [22]. It provides an example of a diagram using the OMT notation for Object Models (Section 7.1) and templates for object, relationship and role binding (Section 7.2 to Section 7.4). All templates are written in quasi-GDMO-GRM.

7.1 Object Diagram

A subnetwork connection is always established between two Network Connection Termination Points (NWCTP) and a trail between two Network Trail Termination Points (NWTTP). The LTP terminates Topological Links (TL).

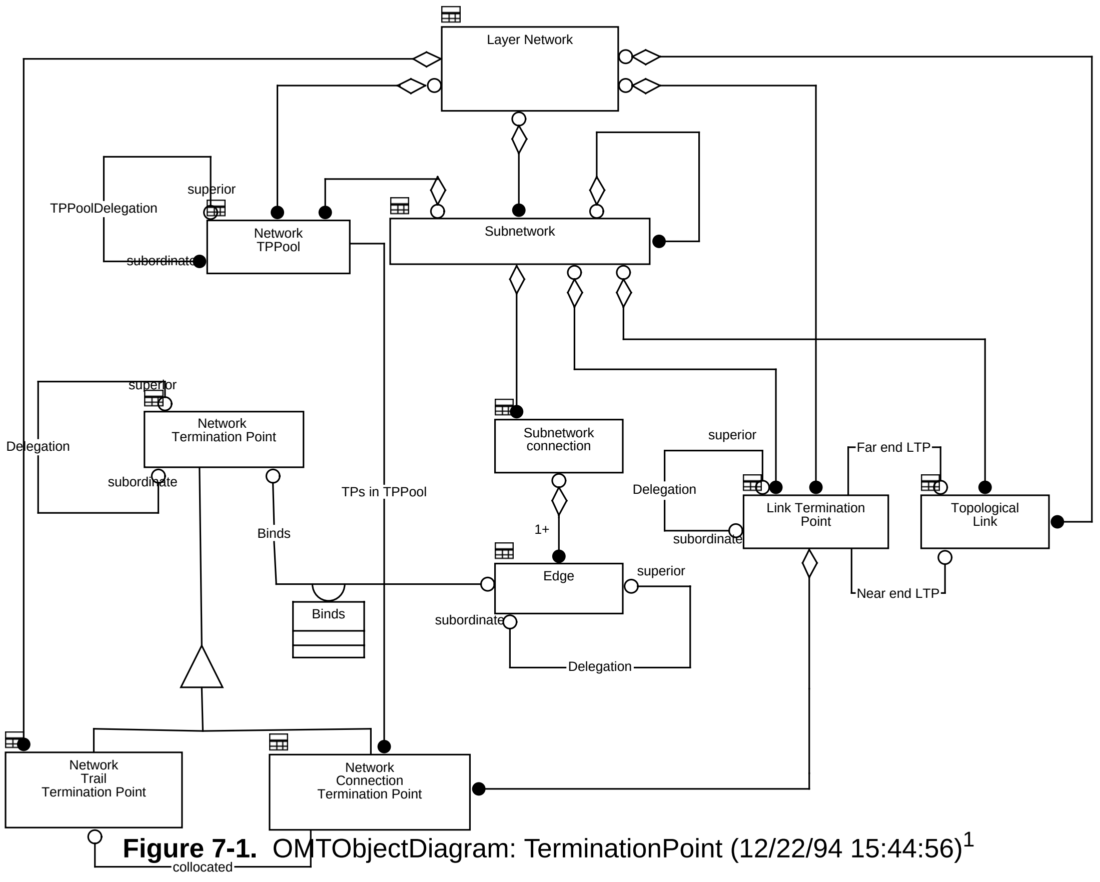


Figure 7-1. OMTObjectDiagram: TerminationPoint (12/22/94 15:44:56)[1]

1. Ignore the small table display marks in figure.

7.2 Object Types

7.2.1 Edge

```
Edge OBJECT TYPE
  DERIVED FROM Configurable;
  CHARACTERIZED BY Edge-package PACKAGE
  BEHAVIOUR Edge-Behaviour BEHAVIOUR DEFINED AS
      "
      COMMENTS: This class binds an NWTP (NWTTP or NWCTP) to a subnetwork
      connection (SNC), trail or a tandem connection (TC). Instances of the
      edge class are created by connection management during setup of con-
      nections and deleted when the related connections are released. Edges
      are dynamically added to and removed from the superior SNC, trail or
      TC during the life time of that SNC, trail or TC. The edge object
      contains information about the role played by the NWTPs (i.e. NWCTP,
      NWTTP), participating in the subnetwork connection. When the SNC,
      trail or TC is created, one edge object will also be created, and it
      will be bound to the source NWTP. The binding procedure between an
      edge and a NWTP is not any requirement during the creation phase. This
      edge is called the root edge (bound to source NWTPs), and it exists
      until the subnetwork connection, trail or TC is deleted. During the
      life time of the subnetwork connection, trail or TC, leaf edges are
      dynamically created and deleted. Leaf edges are bound to sink NWTPs.;
      INVARIANTS: If an edge is an component of a trail, it can only be bound
      to a NWTTP and if an edge is an component of a TC or a SNC it can only
      be bound to a NWCTP. An edge object must and can only be contained in
      one and only one of the following objects: Trail, SNC or TC.;
      create-Edge(x : ANY):(y : Edge)
        PRECONDS: x is of type Subnetworkconnection, Trail, TandemConnec-
        tion, x.administrativeState = unlocked;;
        POSTCONDS: y is contained in x, if there exists z of type Edge con-
        tained in x different from y, then x.rootLeaf = leaf else root, y.us-
        ageState = idle, y.operationalState =  enabled;;
      delete-Edge(x : Edge):(y : ANY)
        PRECONDS: x is contained in y, yis of type Subnetwork Connection,
        Trail, or Tandem Connection, y.operationalState = enabled, y.admin-
        istrativeState = unlocked;;
      hold
        Indicates if the information flow is inhibited. Value is set sepa-
        rately for receive direction and transmit direction.;
      hold-GET(): (x: Boolean)
      hold-REPLACE(x: Boolean): ()
      echo
        Indicates if the information transmitted from the termination point
        is looped back to it.;
      echo-GET(): (x: Boolean)
      echo-REPLACE(x: Boolean): ()
      rootLeaf
      This attribute indicate if this object is a root or leafe edge. The
      first edge attached to e.g. a SNC is named as a root and the rest
      (in case of a multipoint connection) are called leafs.
      This attribute will be set when the edge is created.;
    rootLeaf-GET(): (x: RootLeaf_t)
    usageState
      The usage state might either be idle, busy or reserved.;
    usageState-GET(): (x: UsageState_t)
    usageState-REPLACE(x: UsageState_t): ()
    activate ()
      When an edge is attached to a trail, SNC or TC the state might be
      reserved for the edge. The operation activate change state from re-
      served to busy for a specific edge.;
      PRECONDS: usageState = reserved
      POSTCONDS: usageState = busy
    deactivate ()
      Change state for an edge, from busy to reserved. This operation will
      not be used by the connection management part, its use is for further
      studies.;
      PRECONDS: usageState = busy
      POSTCONDS: usageState = reserved
    ";
ATTRIBUTES
    hold
      PERMITTED VALUES: Boolean
      INITIAL VALUE: False
      GET-REPLACE,
    echo
      PERMITTED VALUES: Boolean
      INITIAL VALUE: False
      GET-REPLACE,
    rootLeaf
      PERMITTED VALUES: RootLeaf_t
      INITIAL VALUE: Leaf
      GET,
    usageState
      PERMITTED VALUES: UsageState_t
      INITIAL VALUE: idle
      GET-REPLACE;
ACTIONS
    activate (),
    deactivate ();
NOTIFICATIONS, ;
```

```
REGISTERED AS ??;
```

7.3 Relationship Types

7.3.1 Binds[Edge-role, NetworkTerminationPoint-role]

```
Binds[Edge-role, NetworkTerminationPoint-role] RELATIONSHIP TYPE
  CHARACTERIZED BY Binds[Edge-role, NetworkTerminationPoint-role]-package
  PACKAGE
  BEHAVIOUR Binds[Edge-role, NetworkTerminationPoint-role]-Behaviour BE-
  HAVIOUR DEFINED AS
      "
      COMMENTS: Identifies the corresponding NWTTP or NWCTP of an edge.;
        create-Binds(x0 : Edge-role, x1 : NetworkTerminationPoint-role):(y
        : Binds[Edge-role, NetworkTerminationPoint-role])
            POSTCONDS: x0.usageState = busy or reserved
        delete-Binds(x : Binds[Edge-role, NetworkTerminationPoint-role]):()
            POSTCONDS: x.NetworkTerminationPoint-role.usageState = idle,
        create-Edge-role():(x : Binds[Edge-role, NetworkTerminationPoint-
        role], y : Edge-role)
            PRECONDS:
        delete-Edge-role(x : Binds[Edge-role, NetworkTerminationPoint-
        role], y : Edge-role):()
            PRECONDS:
      ";
  ATTRIBUTES;
  ACTIONS;
  NOTIFICATIONS;
  ROLE Edge-role
      ROLE CARDINALITY CONSTRAINT (0..1);
  ROLE NetworkTerminationPoint-role
      ROLE CARDINALITY CONSTRAINT (0..1);
REGISTERED AS ??;
```

7.4 Role Bindings

7.4.1 Binds[Edge, NetworkTerminationPoint]

```
Binds[Edge, NetworkTerminationPoint] ROLE BINDING
  RELATIONSHIP TYPE Binds[Edge-role, NetworkTerminationPoint-role]
      ROLE Edge-role
        RELATED TYPES Edge;,
      ROLE NetworkTerminationPoint-role
        RELATED TYPES NetworkTerminationPoint;
REGISTERED AS ??;
```

7.4.2 composition-ExactlyOne-Many[LayerNetwork, NetworkTrailTerminationPoint]

```
composition-ExactlyOne-Many[LayerNetwork, NetworkTrailTerminationPoint]
ROLE BINDING
  RELATIONSHIP TYPE composition-ExactlyOne-Many[composite, component]
      ROLE composite
        RELATED TYPES LayerNetwork,
      ROLE component
        RELATED TYPES NetworkTrailTerminationPoint;
REGISTERED AS ??;
```

8. For Further Study

One of the main reasons for not prescribing formal information specifications is the skills that formal specifications require. However, during the writing of this document the authors had formal specifications constantly in mind. This is reflected in the set of concepts that are introduced. For example, the information specification developers are forced to think in terms of pre-conditions and post-conditions whenever they specify possible changes. This ensures a certain quality of the specifications and will make the translation into a formal specification possible without too much trouble. There are ongoing activities in other international fora (e.g., ITU SG 15) that look at possible formal approaches. Alignment with their results is a topic for further study.

9. Acknowledgments

We would like to thank Martin Chapman, N. Natarajan, Haim Kilov and Jean-Bernard Stefani for valuable discussions, ideas and comments.

We would also like to thank the reviewers and multiple other people from organizations all over the world for their valuable comments.

Erik A. Colban
Telenor
Norway

Heine Christensen
TeleDanmark
Denmark

Appendix A: A Comparison of Different Approaches

A.1 Information Modelling in ODP

For every viewpoint in ODP, there is a corresponding language for the specification of a model in that viewpoint. The Information Language is poorly defined in ODP. Apart from the basic concepts, to be used in all viewpoint specifications, ODP introduces few additional concepts: Invariant Schema, Dynamic Schema, and Static Schema seem to be the most central, see ODP Part 3 ([11]), pp 8--9. Relationships are briefly mentioned. All these concepts are tersely defined, and it is difficult to understand what exactly they should cover and how they should be used. Some intuitive understanding can be obtained by looking at the examples provided in [13] ([ODP Trader]).

The Invariant Schema “defines the semantically consistent structures in an information object (an information object is defined to be an information specification?!) that are independent of any behavior the object might exhibit at any point in time.” [11], p. 8.

The examples of invariant schemata in [13] (ODP-Trader) specify general informal descriptions of the types, and how instances are grouped to form classes of objects (i.e., containment).

The Dynamic Schema is “an action template that defines behavior in which an information object can participate subject to the constraints of an invariant schema” [11], p. 8.

The examples of dynamic schemata in[13] (ODP-Trader), consists of rules that constrain the deletion, creation, classification, reclassification, and declassification of objects.

The Static Schema “defines the state and structure of an information object at some point in time”

No examples of static schemata are found in [13].

No graphical presentation form are provided by ODP, examples of specifications are given in [12] (RM-ODP, Architectural Semantics).

A.1.1 Conclusion:

Information modelling, as prescribed by ODP, needs to be made more precise, especially the schema and other concepts that can be used in connection with the specification of schemata. ODP gives lots of room to adjust other approaches to fit within the ODP framework.

A.2 OSI Management Information Model

The kinds of cross-object relationships that are covered by OSI-MIM are

- classification using classes

- the inheritance relationship

- the manager/managed object relationship

- the Containment relationship - the Naming Tree (Managed Information Tree)

In the OSI MIM, classes can be understood more or less in the same way as template types in ODP. ODP covers probably this concept sufficiently as is.

Inheritance corresponds to inheritance in ODP. Again ODP covers this concept sufficiently on its own. Inheritance is a highly discussed subject in object-orientation forums, and the understanding may vary from author to author or from standard to standard.

The implicit manager/managed object relationship in ISO Management clearly has an information modelling aspect to it that is not covered in ODP, but which could be described, e.g., in some schema. The manager object may control the behavior of the managed object.

The containment relationship, which is a specialization of the composition relationship[1], is probably one of the more important ones in information modelling. Its main purpose is to use it for naming; the distinguished name of an object is obtained by concatenating the distinguished name of its superior object and its relative distinguished name.This relationship is however very rigid in ISO MIM, due to the fact that every managed object is contained in one and only one containing managed object.

The containment relationship is useful in virtue of what it can be used to specify, which in ISO MIM is:

- “define the static behavior of containing and contained managed objects (e.g., constraints on the number and class of managed objects that can be contained in the managed object).”

- “define the dynamic behavior of the containing and contained managed objects (e.g., constraints on the attributes values in containing and contained managed objects, availability of the containing and contained managed objects).”

The naming tree is used specifically for naming purposes and organizes objects in a superior/subordinate relationship, where the superior may be used as part of the name of the inferior. It is a relationship between objects. How this relation relates to the classes of the objects is specified through so-called name bindings.

A.3 OSI General Relationship Model

This model is described only in draft form([20]), but introduces many useful concepts that can be used in defining general relations between objects or between classes (types).

It is concerned with

1. In the case of composition, an instance of the subordinate type (class) may be a component object in zero, one or more instances of the superior (type) class, whereas in the case of containment, an instance of the subordinate type (class) is a component object in exactly one instances of the superior (type) class.

- the specification of “generic relationships”,
- “the specification of the properties of a relationship in terms of roles, cardinalities, entry/departure rules, and overall behavior, independent of how this relationship is represented in managed object class definitions”
- “the specification of the characteristics that must be present in the managed objects participating in the relationship”

Note: Hereafter, the qualifier “managed” will be frequently omitted in front of “relationship” and “object”.

A relationship is a binding between objects, where the characteristics of at least one of the participating objects are affected by at least one of the other participating objects. Note that a relationship is defined to be a binding between objects and not classes (types).

A relationship is defined by a relationship class template.

A relationship class is defined in terms of Roles and Behavior[2].

Each object participating in a relationship fulfills one or more roles.

Note: Roles are not classes, but may be bound to classes by a role binding, see below.

Characteristics of a role are:

- the role cardinality, i.e., the number of objects that may participate in a relationship in that role.
- dynamic or static entry, i.e., whether an object may be bound in a relationship in that role after the establishment of the relationship.
- dynamic or static departure, i.e., whether an object may cease participating in a relationship in that role before the relationship is terminated.

Characteristics of behavior are

- the effects of adding, removing and changing participants in a relationship
- the effects of changes in one or more participants on the other participants including the consistency constraints that must be maintained among the participants
- pre-conditions, post-conditions for (management) operations on the relationship
- invariants

Note that the characteristics of the roles (role cardinality, relationship cardinality, etc.) could alternatively be expressed in the behaviour part of the relationship template.

2. Behaviour has a different interpretation here than in ODP, meaning more or less the behaviour that is not captured by the remaining parts of the relationship specification.

Relationship classes are defined independently of classes. An object may participate in relationships of different relationship classes in different roles, and in more than one role in the same relationship. How roles relate to classes is determined by a Role Binding. A role binding specifies for each role the class of the objects that may participate in that role. It may specify further constraints on the role/relationship cardinality of a given role, and further constraints on behavior in order to maintain “relationship integrity”.

A relationship is integral if no conflict arises between the relationship as reflected by the participants and the system, the specification of the relationship class, and the specification of the role binding.

Generic relationships can be specified; the roles can be seen as type parameters and specific relationships are created by binding specific classes (ODP: object types) to those parameters.

A.3.1 Differences between GRM and TINA

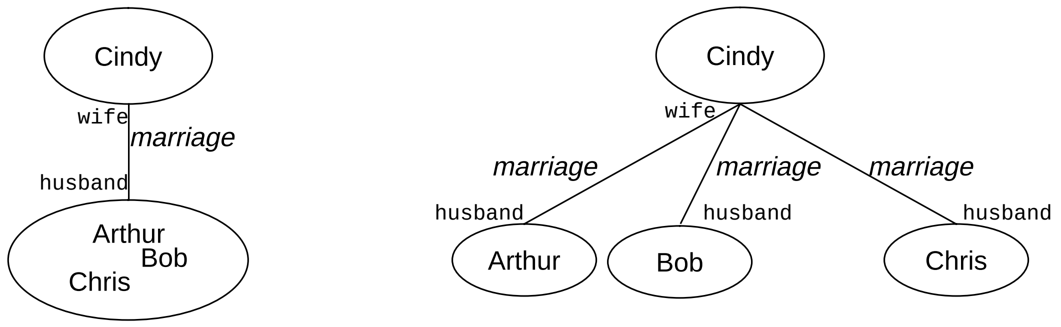
The concepts introduced in Section 4 are very similar to the information modelling concepts of GRM, but there are some differences. This section points out these differences.

By stating that each role of a relationship must be occupied by exactly one object, TINA differs from GRM. In GRM, a *set of* objects may participate in a role in a relationship. In some cases, GRM may seem to allow more flexible ways of modelling relationships.

To exemplify the differences between TINA and GRM, consider first the modelling of the *marriage* relationship type in a polygamic society where women can have several husbands but men can have at most one wife. In GRM, this relationship type could be modelled such that an instance would have one woman in the role of wife and the set of men that the woman is married to in the role of husband. In TINA, exactly one object is bound to each role, so that if a woman is married to more than one man, we would have to model this by a set of relationships; one for each husband, see Figure A-1. Note that the TINA way of modeling this situation is compatible with GRM.

The difference between having a single object and having a set of objects bound to a role, is crucial when defining relationship types that have attributes. (GRM relationships do not have attributes, but a way of working around this is to introduce a relationship object and let the relationship attributes be attributes of the relationship object.) When allowing a set of objects to be bound to a role, a relationship attribute may relate to tuples of *sets of* objects, whereas with TINA’s interpretation of a relationship, a relationship attribute relates to a tuple of single objects. For instance, when modelling the marriage between Cindy and Arthur, Bob, and Chris with one relationship, a relationship attribute would pertain to all four participants of the relationship, whereas in the TINA way of modelling, a relationship attribute would pertain to only the two participants of the relationship it is part of. Supposing that the three men enter the marriage at different times, an attribute such as *Date_Of_Marriage* will only make sense in the second model.

A similar observation holds for actions of relationship types.



Modelling a polygamous marriage with Cindy as wife and Arthur, Bob and, Chris as husbands. The lines are relationships, the ovals are roles. To the left, a possible way of modelling the situation in a way that is consistent with GRM, to the right the situation is modelled in a way that is consistent with TINA.

Figure A-1. Modelling relationships in TINA and GRM

The main reason for only allowing single objects, and not sets of objects, to participate in a role of a relationship, is that becomes possible to redefine *role cardinality* to make it more expressive and useful (see Section 4.4). GRM defines the role cardinality to be the number of objects bound in a relationship in a given role. With this definition, the role cardinality of any role in any relationship is 1 if the TINA interpretation of relationship is used. Role cardinality is redefined by TINA to be a constraint on the number of relationships an object of a given type may be bound to in a given role.[3] For instance, if we want to rule out the possibility of having women married to more than one man, we need to express 1) that at most one man can be bound to the husband role in a given marriage relationship *and* 2) that every woman may be in the role of wife in at most one marriage relationship. The first part of this conjunction is always true under the TINA interpretation of relationships, the second part of the conjunction is expressed by specifying the role cardinality of husband to {0,1} when defining the relationship type.

A.4 Bellcore's Information Modelling

Bellcore's “The Materials: A Generic Object Library for Analysis”, [2], defines several means to specify the relationship between the information contents of several objects. It uses a sort of Entity-Relation approach to specify these relations.

An entity is regarded as an object type, together with rules that govern the behavior of its instances in relation to instances of other associated entities.

Entities are part of associations and their behavior is defined by so-called CRUD rules (Create, Read, Update, Delete)

3. In earlier versions of GRM, there was a concept called *relationship cardinality* that could be used for this purpose. For some reason, this concept has been withdrawn from later versions.

Different kinds of generic associations and entities are identified:

Table A-1. Associations and entities in Bellcore's approach

Association	Entities
Dependency	Parent-Dependent
Reference Association	Reference-Maintained
Composition	Composite-Component
Subtyping	Supertype-Subtype
Relationship Association	Relationship-Regular Entity

This set of associations is extendible; examples show how this associations are defined are given in [20] (using the GRM) and in [5] (using Object-Z).

Rules define the CRUD operations of an association. For every association invariants and CRUD rules for its entities are identified.

A regular entity may have no mandatory participation in any association in which case the CRUD rules are specified by its reference entity as preconditions.

A relationship entity is an entity that associates several "equal" entities (equal in the sense that they obey the same CRUD rules). Example: A shipment is a relationship entity that associates a supplier entity and a part entity.

Among the rules that govern a relationship entity we find, e.g.,:

- A relationship entity instance may exist only if instances of its immediate participating entities exists.

A dependent entity is an entity of which instances may exist only if instances of another associated entity exists. Example: An account entity is dependent on a customer entity; an account can only exist if there exists an associated customer.

Among the rules that govern a dependent entity we find, e.g.,:

- An instance of a dependent entity may exist only if at least one instance of its parent entity exists.

A reference entity is a "read-only" entity that defines pre-conditions for the CRUD rules of another entity (its maintained entity).

Among the rules that govern a reference entity we find, e.g.,:

- Reference entities may be read - but not created, updated, or deleted - by CRUD operations applicable to its maintained entity.

A composite entity “consists of” component entities. Operations applied to a composite entity often propagates to its component entities. Contrary to the OSI Management Information Model, composite entities may share a component entity. The composition association has several subtypes (or refinements), one of them is containment, which corresponds more or less to the OSI MIM containment relationship.

Example: A document may consist of text, pictures, and tables.

Rules govern whether or not component instances may be added or subtracted from a composite.

A subtype entity is one that inherits all properties from some supertype entity. The use of exhaustive subtyping is recommended, which means that every instance of some supertype entity, is always an instance of some subtype of that entity.

Example: A document component has the following subtypes: text, picture, or table. If this subtyping is exhaustive, then every component must either be text, picture, or table.

A.5 Object Modelling in OMT

Although [6] is a book that describes a methodology for object-oriented design, Chapters 3 and 4 cover much of what can be seen as the description of the invariant schemata of ODP.

Object models describe the static data structure of objects, classes (= types), and their relationships to one another.

Two important concepts are introduced, links and associations.

A link is a physical or conceptual connection between object instances, whereas an association describes a group of links with common structure and common semantics. (Compare with relationship vs. relationship class of OSI General Relationship Model.)

Associations and links are described using diagrams. An association refers to classes whereas links refer to object instances. Associations and links are often binary and the authors recommend to keep them simple and not to let them have too high “arity”.

Additional constructs are link attributes, roles, qualifiers.

A link attribute is a property of the links of an association, in the same way as an object attribute is a property of an instance of a class.

A role is one end of an association, e.g., a binary association has two roles, a ternary association has three, etc. Because objects of the same class can play multiple roles in a link, association diagrams are annotated with role names.

Qualifiers are special attributes used to identify subsets of objects that play a given role in a link. Associations often impose multiplicity constraints, i.e., constraints on the number of objects that can play a given role in a link. Qualifiers are used to reduce the multiplicity of a role.

Generic associations that are treated in OMT are aggregations (corresponding more or less to containment in OSI, or composite entities in Bellcore's approach) and generalizations (subtype/supertype relationship in ODP and Bellcore, inheritance in OSI).

A.6 Assessment of the Different Approaches

Table A-2. Concepts Correspondences

ODP	OSI-GRM	Bellcore	OMT	TINA-C
Invariant schema	Relationship class + role binding (invariant parts only)	Association definition including invariants	Object models/ Associations	Invariants in object and relationship templates
Dynamic schema	Non-invariant part of behavior	CRUD rules	Dynamic Models	Operations, notifications, and internal actions specifications
Type	Roles	Entities in the generic associations.	Roles	Type, Roles
Template type	Class	Generic associations and their entities	Class	Template Type
?	Role binding	Instantiation of generic parameters/entities	?	Role Binding
Relationship	Relationship class	Association	Association	Relationship Type
Liaison[a], (tuple of objects)	Relationship	Association instance	Link	Relationship
?	Role cardinality	Role cardinality	Multiplicity	Role Cardinality
?	Relationship object	Relationship entity	Link attributes	Relationship
create	create	create	?	create
delete	delete	delete	?	delete

Table A-2. Concepts Correspondences

ODP	OSI-GRM	Bellcore	OMT	TINA-C
de-, re-, classify	?	dynamic typing	?	?
?[b]	entry/departure	defined through operations	?	creation/deletion of relationship

a. See [RM-ODP], Part 2, p. 11.
b. Entry/departure rules could be specified as part of the establishing or enabled behavior related to a liaison, see [RM-ODP], Part 2, p. 11.

ODP is clearly not prescriptive enough when it comes to information modelling. It only specifies a certain division of the specification into three parts: invariant, dynamic, and static schema.

Useful concepts for information modelling that OSI Management information model provides have already more or less been incorporated into ODP. The distinction between operation and notifications is important in specifying the object types. In order to specify relationships between object types or objects, OSI MIM provides few enhancements.

OSI General Relationship Model provides the richest set of concepts that makes it possible to define different kinds of relationships. The separation between relationship and relationship class, as well as the separation between role and class provides great flexibility and possibilities for reuse. A possible enhancement is to add relationship attributes to the relation classes, corresponding to the link attributes as in Rumbaugh’s approach. Graphical notation may be borrowed from Bellcore's approach or [Rumbaugh]. The dynamic aspects of behavior should be separated from the invariant part, so that a split in accordance with ODP’s dynamic/invariant schema separation is achieved.

Bellcore's approach clearly enhances ODP, and could possibly also be adapted to ODP. The set of generic associations already provided is extendible using various notations. Although Bellcore’s approach does not provide a set of concepts as rich as GRM does (such as relationship cardinality) there is essentially no difference in expressive power.

OMT provides a good rationale for information modelling and is therefore recommended reading. It provides a good graphical notation and covers almost all the concepts that are found in [20] plus some more (e.g., link attributes and qualifiers). Role bindings and relationship cardinality (see [OSI-GRM]) are not covered. Specification of associations could be seen as invariant schemata in ODP, link specifications could be seen as static schemata. Dynamic schemata could be specified as additional rules that refer to the association specification.

Appendix B: Evaluation of Notations

B.1 Requirements on a Notation

B.1.1 Introduction

This section describes requirements on a notation for the information specification. The requirements are divided into:

- General requirements on a notation
- Requirements common to objects and relationships
- Requirements on a notation for objects
- Requirements on a notation for relationships

The requirements are not given in order of significance. The codes G,C,O and R represent the words General, Common, Object and Relationship.

B.1.2 General Requirements

- G1: The notation should be able to precisely express the semantics of an information specification so that all relevant questions about the specification can be answered without referring to outside sources.
- G2:The information specification notation must be compatible with the computational specification notation.

  This does not necessarily mean the two models should use the same notation, however, if desired, a mapping between the concepts of an information specification and a computational specification should be possible.
- G3: The notation must be well-documented and agreed on.

  Standardized notations are suitable but other notations may also be of interest.
- G4: The notation must be publicly available.
- G5: There must be tool support for the notation, e.g. editing, analyzing and if possible animation, simulation.
- G6: The notation must have a formally defined semantics.

  To ensure that the semantics is not ambiguous, it must be formally defined.
- G7: The notation must be or is becoming familiar to the telecommunications and computing community.

B.1.3 Requirements Common to Objects and Relationships

- C1: The notation must describe encapsulated objects, i.e. objects with state and operations (including internal actions and notifications) where only the local operations can work on the state of the object.

  This is a basic requirement of object-orientation.

- C2: The notation must allow to describe object types and relationship types (classification, types).

  Classification is one of the most used forms of abstraction and enables reuse.

- C3: Localization: it must be possible to define an entity (class, object) local to another entity, such that the scope of the enclosed entity is the enclosing entity. Localization defines non-global scopes and makes maintenance and conceptualization easier by defining a local scope of an object.

- C4: The notation must include a vocabulary and a language to express various data types such as integers, sets, booleans, etc.

  Any object must define a state and this has to be structured by attributes with data types.

- C5: The notation must support inheritance between types or classes. Multiple inheritance should be supported for modeling purposes.

  Inheritance is used to model subtyping or specialization. Multiple inheritance is usually used for uniting orthogonal aspects. It is not yet decided whether multiple inheritance is the right answer for this problem. Multiple inheritance brings the problem of name collisions for super types.

- C6: The notation must support specification of dependent creation and deletion of objects.

  For instance a relation may specify that one object's existence depend on another (parent-) object's existence. Then the dependent object is created and deleted with the parent object.

- C7: Invariant Schema: The notation must make it possible to specify invariants of both relationship types and object types (i.e. invariants on several objects participating in a relationship and invariants on the individual objects).

- C8: The notation must make it possible to specify Quality of Service attributes.

  Quality of service may be given on initialization and then kept constant (e.g. formal parameters to a type) and may have special operations to inspect them.

- C9: Dynamic Schema: The notation must make it possible to specify operations and their behavior. The behavior specification must include both the state change of the object to which the operation belongs to as well as effects on other related objects.

  We intend to cover both behaviour of objects and behaviour of relationships.

- C10: The notation must make it possible to specify and reason about specializations of behavior and invariants. For example an invariant I_1 is a specialization of another invariant I_2 if and only if I_1 logically implies I_2.

  Types model concepts and inheritance models specialization between these concepts. Therefore it is necessary to reason about whether the inheritance is a specialization or not. This is also important for inclusion polymorphism where a subtype-object should be able to substitute where a supertype-object is expected.

B.1.4 Requirements on a Notation for Objects

- O1: The notation should facilitate the specification of sequencing control. Sequencing control is the specification of the allowable order (concurrent, sequential or otherwise) of operation invocations.

  It is not yet agreed whether this is a computational requirement or an information model requirement. Sequencing defines the legal order of operation invocations. With a parallel to linguistics, the operations define the alphabet of the object's language and the sequencing control defines the language of the object over this alphabet.
- O2: The notation should support inclusion polymorphism; i.e. the ability to perceive an object of a subtype as being an object of a supertype.

  Inclusion polymorphism is vital to reuse of object specifications.

B.1.5 Requirements on a Notation for Relationships

- R1: The relationship of composition of objects must be provided by the notation. This is a minimal requirement on the kinds of relationships that the notation must support.

  Composition is a generic relationship that must be supported
- R2: In general, relationships (relationship-classes and -objects) and roles (an object participating in a relationship have a role in that relationship) must be provided by the notation.

  Objects should be able to query and invoke relationships, so their type systems should be inter-working (or, the same).
- R3: It must be possible to specify binary, ternary and general n-ary relationships.

  These kinds of relationships arise in modeling the complexity of the real world.
- R4: It must be possible to specify generic relationship types.

  Generic relationship types are important for reuse.

B.2 Evaluation of the GDMO and GRM Notations

B.2.1 Introduction

In this section we evaluate the use of the notations prescribed in [19] (GDMO) and [20] (GRM). The GRM notation is an extension of the GDMO notation, and it is therefore natural to evaluate these together. Note that we are only evaluating the notations in order to see how they can be applied to provide information specifications in accordance with the information modelling concepts described in this document and that we do not commit to the underlying Management Information Model.

B.2.2 Evaluation against General Requirements

- G1: Parts of the definition of managed object classes (GDMO) and relationship classes (GRM) are defined in a so-called behavior field, that can contain any text. In order to write complete specifications, GDMO and GRM need to be extended with some notation to use in the behavior.
- G2:GDMO is a candidate for the computational specification notation.
- G3: GDMO is an ISO Draft International Standard, GRM is a Committee Draft.
- G4: Both are publicly available.
- G5:Tools exist for GDMO, although they may not be publicly available. Both notations are based on ASN.1 and ASN.1 syntax checkers and compilers could be used as a tool.
- G6: Neither notation has a formally defined semantics.
- G7: GDMO is commonly used in the telecommunications industry by people involved in network management. GRM is still too fresh to have gained any popularity, yet. The recommendations are still in draft form and undergoing changes.

B.2.3 Evaluation against Requirements Common to Objects and Relationships

- C1:GDMO allows one to describe encapsulated objects. However, it is often assumed in the OSI information models that the state of a managed object may change independently of any operation invocation. This may, e.g, happen if a change occurs in the resource that the object represents. The change may occur as the result of a proprietary change in the resource. In order to account for encapsulation GDMO must be used in a much stricter manner, explicitly expressing every possible state change by defining an operation for each change of state.
- C2:Both GDMO and GRM are used to specify object classes and relationship classes.
- C3:This is not supported by GDMO and GRM. Actually, the object identifiers have global scope and each has a specific place in a tree. It is not possible to define managed object classes or relationship classes within managed object classes or relationship classes.
- C4: The ASN.1 notation may be used for this purpose.
- C5: Multiple inheritance is supported by both GDMO and GRM.
- C6: The notation must support specification of creation and deletion of objects.

  Managed objects may be created and deleted by managed protocol interactions.
- C7:Invariants are described in the behavior part of the templates. The behavior templates are strings where one can write the specification usually in English.
- C8:Any attribute can be used for that purpose, however, it is not clear what kind of property should be given to such attributes (`GET`, `GET-REPLACE`,?).

Sometimes the value of a Quality of Service attribute may change, but cannot be changed unreservedly.

- C9:See remarks for C7 (Invariant Schema) above.
- C10:Specialization is realized in GDMO through inheritance, using the `DERIVED FROM` construct. The process of specialization requires that all “characteristics” of the superclass(es) to be included in the definition of the subclass. The characteristics are not necessarily described and it therefore becomes difficult to reason about behavior.

B.2.4 Evaluation against Requirements on a Notation for Objects

- O1: This could be specified in the behavior, for which GDMO provides no structure.
- O2: The way that GDMO has defined inheritance, inclusion polymorphism is supported.

B.2.5 Evaluation against Requirements on a Notation for Relationships

- R1:The composition relationship can be specified with GRM.
- R2:GRM provides these concepts.
- R3: GRM allows one to specify relationships of any “arity”.
- R4: GRM allows one to specify generic relationship types.

B.2.6 Conclusion

The lack of formal behavior specification is one of the major drawbacks of GDMO and GRM. The possibility to extend the notations with formal specifications of behavior exists, but it is unclear exactly how this should be done. If formal specifications of behavior is requested it seems to be easier to only apply one of the already existing formal description techniques (FDT), e.g., Object-Z, rather than extending GDMO and GRM. The intermix of the complexity of GDMO and GRM together with the complexity of an FDT will make it even more difficult for the users to read and write specifications.

Another drawback with GDMO and GRM is that they traditionally have been used in a very specific way (OSI Network Management) and it will require a certain shift of paradigm for users to apply it to general information specifications. As shown in Section 6, special care must be taken in order that defined objects meet the encapsulation property.

Therefore, although the GDMO and GRM notations meet most of the requirements above, the traditional use of these notations are not necessarily consistent with principles of object-orientation discussed and presented in this document. If GDMO+GRM is used as a notation, guidelines for its use will have to be provided.

B.3 Evaluation of Object-Z

B.3.1 Introduction

Z and Object-Z are abstract mathematical notations building on set-theory and first order logic. Object-Z is an object-oriented extension of Z and since some of the requirements relate to object-orientation we only evaluate Object-Z.

B.3.2 Evaluation against General Requirements

- G1: Object-Z is capable of expressing both objects and relationships in a concise, precise manner.
- G2: Object-Z could also be used for specifying abstract computational specifications. The ground work of object-orientation provides the basic mapping mechanism between information model specifications and computational model specifications.
- G3: Z has a british standard and Object-Z is described in a report[15].
- G4: Object-Z is publicly available.
- G5: There exist type-checkers for Z and layout tools for Z and Object-Z
- G6: The semantics of Z has been formally defined (in Z itself) in [17]. Object-Z has a semantics given by natural language in [15].
- G7: Z is one the four FDTs recognized by ODP [12].

B.3.3 Evaluation against Requirements Common to Objects and Relationships

- C1: Object-Z supports encapsulated objects with state and operations. Notifications have no particular support in Object-Z.
- C2: Object-Z specifications describe classes which are types of objects. Relationship types can be seen as classes too.
- C3: Object-Z has no particular means of localization beside encapsulation. Components can be declared in the state of an object, however, these are references to global objects.
- C4: Object-Z uses set theory to declare attributes in the state of an object. Means are defined for constructing sets, powersets, etc., sequences, cross products and relations.
- C5: Object-Z supports multiple inheritance. The problem of name collisions (inheriting the same name from different supertypes) is handled by renaming.
- C6: Object-Z provides Init schemata that define properties of the state of an object when it is created. In this schema it is possible to define that other (dependent) objects are created and initialized. It is also possible to specify the dependencies in invariants and operations.
- C7: Classes define state invariants in the state schema. These are defined with first-order logic. As both objects and relationships are modelled with classes, we have class invariants in both.

- C8: Classes can define constant attributes in Object-Z. These attributes can participate in invariants and define quality of service properties.
- C9: Object-Z classes define operations on the state of the object.
- C10: First order logic allows reasoning about implication between assertions in schemata. This makes it possible to reason about whether an invariant implies another and whether an operation is a specialization of another.

B.3.4 Evaluation against Requirements on a Notation for Objects

- O1: Sequencing constraints are described by history expressions in Object-Z. These use temporal logic (logic of time).
- O2: Object-Z supports inclusion polymorphism.

B.3.5 Evaluation against Requirements on a Notation for Relationships

- R1: Only composition using references to the composed objects is possible in Object-Z. This can be regarded as a limitation.
- R2: Both object types and relationship types are represented as classes in Object-Z. This means it is possible for an object to communicate with a relationship.
- R3: Object-Z relationship classes allow description of any kind of relation.
- R4: Object-Z has the possibility of defining generic classes, especially relationship classes.

B.3.6 Conclusion

The evaluation shows that Object-Z is almost ideal for information model specifications.It allows defining classes, relationship classes, communication between them and generic relationships. Furthermore Object-Z is a formal notation and therefore unambiguous (to be absolutely precise you need a formal semantics for a notation to be unambiguous; attempts at a formal semantics of Object-Z have been made [?]).

The problem with Object-Z is the learning curve and mathematical skills required of users of the notation as Object-Z inherits the mathematical vocabulary of Z.

B.4 Evaluation of SDL

B.4.1 Introduction

SDL is the ITU-T standardized Specification and Description Language. It builds on the paradigm of Extended Finite State Machines and asynchronous signalling. SDL has recently (1992) been standardized with an object-oriented extension. SDL is widely used within the telecommunications industry.

B.4.2 Evaluation against General Requirements

- G1: SDL is aimed at expressing objects and communication, not relationships.
- G2: SDL is a viable candidate for a computational notation.
- G3: SDL is standardized by the CCITT recommendation Z.100 [23].
- G4: SDL is publicly available.
- G5: There is full tool support for the SDL88 recommendation; tools are currently being build for SDL92.
- G6: SDL has a formally defined semantics (Annex F of [23])
- G7: SDL is widely used within the telecommunications industry.

B.4.3 Evaluation against Requirements Common to Objects and Relationships

- C1: Processes are encapsulated objects in SDL.
- C2: Systems, blocks and processes can be described by types in SDL.
- C3: Block and process objects are local to their enclosing blocks.
- C4: Processes have local data that are described by ACT-ONE data types along with some build-in data types. SDL is prepared for ASN.1 and C and C++ data types.
- C5: SDL supports single inheritance for all types except timer types.
- C6: SDL has a primitive for creating a process object. A process object can delete itself when requested. Blocks can neither be dynamically created nor deleted.
- C7: There is no explicit way of describing invariants in SDL. The exception is the ACT-ONE data types which describe logical assertions on behaviour of data types.
- C8: SDL has no particular notion of Quality of Service attributes. Process types can be equipped with formal parameters which can describe the service attributes of the particular process instance.
- C9: Operations and behaviour can be defined in two ways in SDL: 1) with signalling where the behaviour is defined by the Extended Finite State Machine (EFSM); 2) with exported procedures where the order of procedure invocation is defined by the EFSM and the behaviour of the procedure is defined by its body.
- C10: Specialization of behaviour can be done with virtual inputs and virtual transitions. This does not require the extension to be a specialization and must be used with care.

B.4.4 Evaluation against Requirements on a Notation for Objects

- O1: The EFSM of a process defines the order of receiving and sending signals as well as the order of exported procedure invocation.
- O2: SDL does not support inclusion polymorphism.

B.4.5 Evaluation against Requirements on a Notation for Relationships

- R1: SDL composes systems out of blocks, blocks out of blocks, blocks out of processes, processes out of services (however, not processes out of processes).
- R2: There is no special means in SDL for defining relationships and roles.
- R3: See R2.
- R4: SDL has the possibility to define generic types, however, see R2

B.4.6 Conclusion

SDL is a reasonable choice for notation for objects; the shortcomings are composition of processes, no multiple inheritance and no behaviour specialization. SDL is not suitable as a notation for defining relationships between objects.

SDL is not as abstract a language as e.g. the FDT Z, and as such more suited to the other abstraction levels of the Work Area A Framework Architecture.

B.5 Evaluation of COOLish

B.5.1 Introduction

ROOM is the object model and COOLish the specification language of the RACE ROSA (Race Open Service Architecture) project. COOLish attempts to define a “pure” object-oriented approach (i.e. not building on an existing non-o-o language).

B.5.2 Evaluation against General Requirements

- G1: COOLish is aimed at expressing objects, not relationships.
- G2: COOLish is being considered a candidate for the computational model notation also.
- G3: COOLish is described by a ROSA report [24] and is not a standard.
- G4: COOLish is publicly available.
- G5: No tool support exists yet for COOLish. An analyzer is currently being build.
- G6: COOLish has not been given a formal semantics; only an informal description exist [].
- G7: COOLish is a new notation developed by the RACE ROSA project.

B.5.3 Evaluation against Requirements Common to Objects and Relationships

C1: COOLish supports encapsulated objects.

- C2: COOLish has the notion of an object type describing the properties of its instances.

- C3: COOLish objects can have local objects. COOLish object types cannot have local object types. COOLish defines scope control by declaring within an object type which other object types are visible.
- C4: COOLish does not prescribe a data notation, but is prepared for ACT-ONE data types.
- C5: COOLish support multiple inheritance.
- C6: COOLish support explicit creation and deletion of objects. Furthermore operations that are automatically invoked when the object is created and deleted can be defined.
- C7: COOLish is an imperative language and as such not particularly developed for expressing invariants.
- C8: There are no particular means for describing Quality of Service in COOLish.
- C9: Behaviour is specified imperatively in COOLish.
- C10: COOLish defines the notion of consistent extensions to reason about specialization of behaviour (and attributes).

B.5.4 Evaluation against Requirements on a Notation for Objects

- O1: COOLish defines the usage clause for sequencing and concurrency control. These are extended regular expressions defining the allowable order of operation invocations.
- O2: COOLish supports inclusion polymorphism.

B.5.5 Evaluation against Requirements on a Notation for Relationships

- R1: Composition of objects is defined by COOLish.
- R2: COOLish is not particularly aimed at expressing general relationships. They would have to be defined as data types consisting of object references.
- R3: see R2
- R4: COOLish has no concept of generic types

B.5.6 Conclusion

COOLish is very suitable as a notation for objects. It provides a “pure” object-oriented approach, specialized for telecommunications. COOLish is not suitable as a notation for relationships.

B.6 Conclusion on the Notation Evaluation

We have listed our requirements on a notation for information modeling specifications and evaluated 4 notations according to the requirements.

Our conclusion suggests that there are two suitable possibilities for the notation:

- GDMO with GRM

- Object-Z

The advantage of GDMO with GRM is that GDMO is widely used and known to the telecommunications management community. There is also a large body of GDMO specifications available for reuse. However, the need for modification of the notation and reinterpretation of some of the key concepts constitute a major drawback of this choice. It may become very difficult to get people to follow the guidelines that will ensure “correct” use in the sense of TINA-C. Another drawback of GDMO is the lack of formal precision.

Object-Z, on the other hand, is an almost ideal notation for information modeling specifications. It expresses objects and relationships within the same language concepts in a concise, precise and formal way that lends itself to intuitive understanding. The drawback of Object-Z is the learning curve and mathematical skills required for users.

Our final conclusion is that a *modified* version of GDMO with GRM is the most suitable candidate for the TINA-C information model specification notation. This is a pragmatic choice where we weigh concerns of learning curve and reusability higher than preciseness and conciseness. In Appendix C, we have included some Object-Z specifications of parts of the example in Section 7.

Appendix C: Example of Object-Z

References

[1] Bellcore , ST-OPT-002008, *The Framework: A Disciplined Approach to Analysis*, Issue 1,May 1992.

[2] Bellcore, ST-OPT-002010, *The Materials: A Generic Object Library for Analysis*, Issue 1, October 1992.

[3] Bellcore, SR-OPT-001826, *Information Modeling Concepts and Guidelines*, Issue 1, January, 1991.

[4] Haim Kilov, *Understand -> Specify -> Reuse: Precise Specification of Behavior and Relationships*, *Proceedings of the IFIP/IEE International Workshop on Distributed Systems: Operation and Management (DSOM '92)*, Munich, Germany, Proceedings pp 1-10, October 12-13, 1992.

[5] Haim Kilov, *Information Modeling and Object-Z: specifying generic reusable associations*, *The International Workshop on Next Generation Technologies and Systems"*, Proceedings pp 182-191, Haifa, Israel, June 28-30, 1993.

[6] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall: Englewood Cliffs, N.J.:, 1991.

[7] ANSA, *The ANSA Reference Manual Release 01.01*, Poseiden HouseCastle Park: Cambridge, UK:, July, 1989.

[8] ISO/IEC JTC1/SC21/WG7, Draft Recommendation X.901, *Basic Reference Model of Open Distributed Processing - Part 1: Overview and Guide to Use,* International Organization for Standardization and International Electrotechnical Committee, June 1993

[9] ISO/IEC JTC1/SC21/WG7 N 7988, Draft Recommendation X.902: *Basic Reference Model of Open Distributed Processing - Part 2: Descriptive Model*, International Organization for Standardization and International Electrotechnical Committee, 25 June 1993.

[10] ANSI, *Comments to Accompany a NO Vote on the Ballot on CD 10746-2.2, Basic Reference Model of Open Distributed Processing, Part 2 -Descriptive Model,* April 3, 1993.

[11] ISO/IEC JTC1/SC21/WG7, Draft Recommendation X.903: *Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model,* International Organization for Standardization and International Electrotechnical Committee, 15 June 1993.

[12] ISO/IEC 10746-5 / CCITT Recommendation X.905, *Basic Reference Model of Open Distributed Processing - Part 5 - Architectural Semantics,* International Organization for Standardization and International Electrotechnical Committee, December 1991.

[13] ISO/IEC JTC1/SC21/WG7 and CCITT SGVII.Q19, *Working Document on Topic 9.1 -*

Electrotechnical Committee, November 1992.

[14] *Computational Modelling Concepts*, Document No. TB_A2.NAT.002_3.0_93, TINA-C, December 1993.

[15] R. P. Duke, G King. Rose, and G. Smith, *The Object-Z Specification Language Version,* Technical report No. 91-1, The University of Queensland, Australia, May 1991.

[16] J. M. Spivey, *The Z Notation, A Reference Manual*. 2nd Edn., Prentice Hall: Englewood Cliffs, N.J., 1992.

[17] J. M. Spivey, *Understanding Z*, Cambridge Tracts in Theoretical Computer Science 3, 1988.

[18] ISO/IEC IS 10165-1 / CCITT Recommendation X.720, *Information Technology - Open Systems Interconnection - Structure of Management Information - Part 1: Management Information Model,* International Organization for Standardization and International Electrotechnical Committee, September 1991.

[19] ISO/IEC DIS 10165-4, CCITT Recommendation X.722, *Information Technology - Open Systems Interconnection - Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects (GDMO),* International Organization for Standardization and International Electrotechnical Committee, September 1991.

[20] ISO/IEC 10165-7 / CCITT Recommendation X.725, *Information Technology- Open Systems Interconnection - Structure of Management Information - Part 7: General Relationship Model,* International Organization for Standardization and International Electrotechnical Committee, January 1993.

[21] *Telecommunications Applications Requirements*, Document No. TB_A0.MH.002_3.0_93, TINA-C, December 1993.

[22] *Network Resource Information Model Specification*, Document No. TB_LSR.011_1.3_94, TINA-C, December 1993.

[23] CCITT Recommendation Z.100 : *CCITT Specification and Description Language (SDL),* March 1992.

[24] RACE: ROSA RACE Deliverable 93/BTL/DS/A/010/b1,*, Foundation of Object Orientation in ROSA, Release Two*, RACE, May 1992.

[25] Jean-Michel Cornilly et al., *Experimenting with ODP Concepts for SDH Transmission Network Management Specification*, *TINA'93 - The Fourth Telecommunications Information Networking Architecture Workshop* (L'Aquila Italy: September 27-30, 1993) Proceedings Vol I pp 67 - 86.

[26] Bertrand Meyer, *Introduction to the Theory Languages,* Prentice Hall, Englewood Cliffs, N.J., 1991.

[27] CCITT Recommendation G.803, *Architectures of Transport Networks Based on the*

Synchronous Digital Hierarchy (SDH), June 1992.

[28] *StP Tool Documentation*, Document No. TP_EAC.004_0.1_93, TINA-C, October1994.

Acronyms

- **ANSA:** Advanced Networked Systems Architecture
- **ASN.1:** Abstract Syntax Notation One
- **CRUD:** Create, Read, Update, Delete
- **CTP:** Connection Termination Point
- **DPE:** Distributed Processing Environment
- **FDT:** Formal Description Technique
- **GDMO:** Guidelines for the Definition of Managed Objects
- **GRM:** General Relationship Model
- **IDL:** Interface Definition Language
- **LTP:** Link Termination Point
- **MIB:** Management Information Base
- **MIT:** Management Information Tree
- **MO:** Managed Object
- **ODP:** Open Distributed Processing
- **OMG:** Object Management Group
- **OMT:** Object Modeling Technique
- **OO:** Object Oriented, Object Orientation
- **OSI:** Open Systems Interconnection
- **OSI-MIM:** OSI Managed Information Model
- **OSI-SMI:** OSI Structure of Management Information
- **RM-ODP:** Reference Model of Open Distributed Processing
- **SDL:** Specification and Description Language
- **TINA:** Telecommunications Information Networking Architecture
- **TINA-C:** TINA Consortium

Glossary

- **Abstraction**: The process of suppressing irrelevant detail to establish a simplified model, or the result of that process. Abstraction consists of *clustering* and *generalization*.
- **Action:** Something that happens. Every action of interest for modelling purposes is associated with at least one object.
- **Allomorphic Behavior**: Allomorphic behavior is exhibited by a *managed object* belonging to one managed object class when it responds to a management operation as if it were an instance of another managed object class.
- **Arity**: The number of arguments taken; unary is one argument, binary is two arguments and n-ary is n arguments.
- **Association**: OMT terminology; see *relationship type*.
- **Attribute**: Information of a particular type concerning an *object*.
- **Attribute type**: An attribute type is that component of an *attribute* which indicates the *type* of information given by that attribute.
- **Behavior (of an object):** A collection of actions with a set of constraints on the circumstances in which they may occur.
- **CRUD rules**: Create, Read, Update and Delete rules for an object given in an *information specification*.
- **Class (of <X>s):** A collection of <X> s satisfying a type. <X> may be any of: object, relationship, interface, action
- **Classification**: is the process of collecting *phenomena* or concepts into a concept.
- **Clustering:** Clustering is the process of collecting phenomena into groups with common properties called concepts. It is the inverse of exemplification.
- **Composition:** 1) (Composition of <X>) An action or the result of realizing a certain <X> from a set of basic or simpler <X>s. <X> may be any of: object, behavior, service. 2) A binary generic relationship type with two roles Composite and Component used to specify relationships between objects were one can be seen as “part of” or contained (see Containment 1)) in the other by letting the former be bound to the Component role and the other to the Composite role. [2]
- **Concept**: A generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection.
- **Containment**: 1) This is a relationship between phenomena. One phenomenon is contained in another phenomenon if it is an intrinsic part of the containing phenomenon. The phenomena are often objects. 2) A generic relationship type which is a subtype of the generic relationship type Composition (see Composition 2))and which has the additional constraint that the role cardinality of the Composite role is {1}.

- **Dependent object/entity**: An object or entity is dependent on another object or entity if and only if its creation or deletion (its existence) depends on the existence of the second object or entity (the parent).
- **Dynamic typing**: The ability for an object to change its membership of object types dynamically during its existence.
- **Exemplification:** The process of delivering an instance of a concept. The inverse process of clustering.
- **Generalization**: Generalization is the process of defining a concept from another concept by removing detail. It is the inverse process of specialization.
- **Generic relationship type**: A generic relationship type defines a number of relationship types by omitting to specify the object types associated with each role. The generic relationship type can be instantiated into a relationship type by specifying the types that are associated with the roles.
- **Information in a system**: The knowledge necessary to make appropriate use of a system.
- **Information modelling concepts**: A collection of *concepts* that provide the framework for the *information specification* of distributed applications.
- **Information object**: An object that occurs in an *information specification*.
- **Information specification**: A description of a structure that models the *information in a system* in terms of information bearing entities, relationships between the entities, and constraints that govern their behavior, including creation and deletion.
- **Information Viewpoint:** One of the ways of looking at a distributed application, as defined in RM-ODP. In this view of the distributed application and its management, the only things visible are the relevant information elements and their relationships.
- **Inheritance**: is a specification technique that defines an *object type* from another object type by adding or overwriting properties (attributes, behavior).
- **Instance**: (of an <X> template), an <X> instantiated from the template. <X> may be any of: object, interface, building block, cluster.
- **Instantiation:** (of an <X> template) A process which using a given template and other necessary information, results in a new <X>. <X> may be any of: object, interface, building block, cluster
- **Invariant**: A property of a system or part of it, valid at all times. An invariant of an *object* is a logical property that holds from after the *object* has been created and initialized and until the object is deleted, except during the execution of operations on the object. Invariants are usually given for *object types*, defining the same invariant for all instances of the object type.
- **Is-a relationship:** the is-a relationship between concepts defines specialization.
- Link: OMT terminology, see *relationship*.
- **Localization**: the ability to define a local scope for a *phenomenon* or *concept*, i.e. a scope constrained to some other concept or phenomenon.

- **Model**: An abstraction of a number of *concepts* or *phenomena* in a system.
- **Object**: a model of a phenomenon.
- **Object template**: See Template.
- **Object type**: See Type.
- **Phenomenon**: Anything that has a definite, individual existence in reality or in the mind.
- **Polymorphism**: The ability to substitute an *object* of one type for an object of a super type in any context.
- **Relationship**: A tuple of objects related by some properties that do not pertain to any particular object in the tuple, but to all objects of the tuple
- **Relationship binding:** The association of the roles of a generic relationship with object or relationship types.
- **Relationship binding template:** The specification of a relationship binding.
- **Relationship type**: A predicate on relationships describing common characteristics of relationships.
- **Role**: A position in a *relationship*. Relationships belonging to the same *relationship type* have the same set of roles. The relationship type may describe properties associated with each role. Each role is associated with one type.
- **Role cardinality**: For a given binary *relationship type*, a set of non-negative integers associated with a role which constrains the number of relationships of the given relationship type that have the same object in the other role.
- **Schema**: A number of properties collected together.
- **Specialization:** Specialization is the process of defining a concept from another concept by adding detail. It is the inverse process of generalization.
- **Subclass:** One class is a subclass of another class precisely when it is a subset of the other class
- **Subtype**: *Type* T_2 is a subtype of type T_1 if and only if (being of type) T_2 implies (being of type) T_1.
- **Template:** The specification of the common features of a collection of <X>s in sufficient detail that an <X> can be instantiated from it. <X> can be any of: object (information object, computational object, engineering object), interface, relationship.
- **Template type**: A predicate of the form "is instantiated from template *t*".
- **Type:** A predicate characterizing <X>. <X> can be any of: object (information object, computational object, engineering object), interface, relationship.