Mondel: An Object-Oriented Specification Language

Gregor v. Bochmann, Michel Barbeau, Mohammed Erradi,
Louis Lecomte, Pierre Mondain-Monval, and Normand Williams

Centre de Recherche Informatique de Montréal (CRIM)
3744, rue Jean-Brillant, Bureau 500
Montréal, Québec, Canada, H3T 1P1

and

Département d'IRO, Université de Montréal
CP 6128, Succursale A
Montréal, Québec, Canada, H3C 3J7

Abstract: In this paper, we present the object-oriented language Mondel. Mondel is an executable specification language with a formally defined semantics. It supports persistency and concurrency. Issues related to the choice of the language features are discussed. We also propose a methodology to guide the designer during the development of a specification. Mondel and the methodology are illustrated by a simple database example. Moreover, we give an overview of the formal semantics of Mondel and explain its use for the development of an execution environment, and for the formal verification of Mondel specifications using techniques from coloured Petri nets. The development of a reflective language definition is also discussed.

1. Introduction

This paper presents a new object-oriented specification language, called Mondel. More importantly, the paper discusses the language concepts which are important for a specification language to be applied in the area of distributed systems. This work was performed in the context of a CRIM-BNR research project for the modelling of the management and operational aspects of communication networks, in particular fault detection and recovery. However, we think that the same principles apply to the specification of other kinds of distributed systems, such as real-time control systems for automated manufacturing and other applications.

A large number of object-oriented languages have been developed recently. Table 1 shows the characteristics of those languages that were particularly considered during the design of Mondel. Also a number of interesting non object-oriented specification languages exist, in particular for the description of OSI communication protocols and services [ISO 87b, ISO 87c, CCITT 87]. We have developed this new language, since we considered that none of the existing languages fitted sufficiently well (1) the requirements for writing system descriptions at the specification and design level, (2) supporting concurrency as required for distributed systems, and (3) being object-oriented with support for persistency.

Much literature exists about the advantages of object-oriented languages (e.g. [Meye 88]) and databases [Gard 87]. In summary, we believe that the following aspects of "object-orientation" are of particular importance:

(1) Humans tend to think in terms of models of the real world in which "objects" are the unit of consideration. It is therefore natural to have objects within the specification of a system to be developed. In addition, a system specification often contains "specified objects" which represent the objects of the real world that are controlled by the specified control system.

(2) Using objects as building blocks for specifications and programs provides for information hiding [Parn 72]. Only a fixed set of operations and functions are externally visible; internal aspects cannot be used for the design of other objects.

(3) The inheritance structure of object classes (found in most object-oriented languages) is a useful concept for the structuring of complex specifications and programs. It also plays an important role for the issues of reusability and extensibility of specifications or parts thereof.

(4) The concept of a class of "persistent" objects corresponds to the notion of entities stored in databases. They have a lifetime of their own, independently of the application programs using them. They often correspond to objects in the real world.

It has often been observed that a language leaves much freedom to the user of the language, and that methodologies for the development of specifications, designs and programs are required to guide the analyst/programmers during their work. Traditional design methodologies are either functionally oriented (e.g. Structured Analysis [Ward 89]) or concerned with the design of databases (e.g. entity-relationship methodology [Chen 76]). We believe that these aspects must be integrated in a methodology and language for developing object-oriented specifications/designs.

Section 2 describes such an object-oriented methodology for system design which is based on similar approaches described in the literature. We have applied this methodology and the Mondel language to the development of a specification of management aspects of communication networks [Boch 90a], of the OSI Reference Model [Mond 90b], and of the OSI Directory system specification [Poir 90]. A simple example is discussed in Section 4 below.

Section 3 presents the major characteristics of the Mondel language and discusses the issues related to the choice of the language features. We try to give justifications for the various design choices which were made for the design of the language.

A specification written in Mondel, like any other system specification, needs to be validated. It also represents the basis for further system refinement or implementation, it is the basis for the development of test cases, and the reference in respect to which the test results, obtained from executing the refined system specification or implemention, are analysed (see e.g. [Boch 90c]). In order to support these different development activities, a specification language should have a formally defined syntax and semantics, and should be executable, as much as possible. Section 5 presents our approach to the formal definition of the semantics of Mondel, presents a simulation system written in

Prolog, and discusses issues related to the validation of specifications. We also discuss a reflective language definition which could provide the theoretical basis for handling dynamic updates of the specification during its execution. This is important for the maintenance of on-line systems.

Several object-oriented programming/specification languages have been proposed in the literature. We believe that Mondel is both original and interesting because of the following aspects:

(1) It is readable and combines many concepts which, to the best of our knowledge, are not found in another single object-oriented language.

(2) It is supported by a methodology for guiding the users in the production of either a global or a detailled design specification.

(3) It is an executable specification language with a formally defined semantics.

2. Design Methodology

This section presents an object-oriented software design methodology. The goal is to provide some simple, efficient, means to structure the process leading from some informal application specification to a formal object-oriented design.This methodology is based on the concept of objects; i.e., that software applications can be designed based on the principle of aggregating data items together with operations performed on them. The application can be seen as a composition of such objects. Advantages of this design approach are numerous though it still stays a very informal, intuitive, process based on human expertise. This methodology was defined [Mond 90a] for the Mondel language, whose main features are described in Section 3. However, we believe it is general enough to be applied to any object-oriented language. Second, this methodology acknowledges the need to relate object-oriented concepts to the more classical, function oriented, design practices [Ward 89, Bois 89]. Thus our methodology uses some Entity/Relationship concepts [Chen 76] already widely applied to software design. Finally, it appears that a generic design methodology can always be adapted for some specific applications areas. Also it seems highly desirable to stay as close as possible to the practices of telecommunication [T1M1 89, ISO 89a, ISO 89b, ISO 89c, ISO 89d], distributed processing [ODP 88], and various other information processing standardization groups [ISO 87a].

Current object-oriented design methods usually comprise a certain number of design phases. Though not all practitioners agree on the number and the denomination of these phases, all come up with approximately the same philosophy [Booc 86, Meye 87, Bail 88, Bail 89, Jalo 89, Bail 90]. We identified four major phases that can be iterated until a satisfactory design is obtained:

(1) A preliminary phase first consists of the identification of the general problem area, of the specific aspects we want to handle, and also on the aim of the expected design; this phase is an informal one, but it should lead the designer to separate the different aspects included in the application, and also to precisely define the intended purposes of the model to be elaborated.

(2) A second phase consists of the identification of the key components of the so-called "application domain"; this phase takes as input any information describing the functionalities to be provided, and should produce as a result a set of entities,

together with relationships among them, that can be easily mapped onto object-oriented languages constructs.

(3) The third phase is concerned with the allocation of the functions to be provided as operations offered by objects identified in the previous phase; some functions will also uncover some new objects which must be integrated in the application domain.

(4) The last phase is concerned with the definition of the behaviors of the objects; these behaviors consider the possible sequences of operation calls as well as the necessary processing associated with each operation; complex objects can be specified as smaller "applications"; i.e., the design process starting in Phase 2 can be applied to each individual object as it is applied to the global application.

The following sections provide some more details on each of the previous phases together with some references to a simple manufacturer database definition example presented in Section 4.

2.1 Phase 1: Problem Definition

This preliminary phase must help the designer to focus on relevant aspects of the general application area. For example, let us consider our first experimentation in the network management area [Boch 90a]. The area of network operation, administration, maintenance, and provisioning (OAMP) is a very complex field due to the intertwining of very different function: data transmission, configuration management, error recovery, service optimization, security control, accounting, and many others. Second, due to the geographical distribution of networks, as well as evolution over time, application specifications must achieve a high-level of abstraction and genericity: the specifications must stay valid for a wide variety of equipment from different manufacturers, as well as for various configurations and services which evolve over time. Complexity is also due to the large variety and number of components: various pieces of equipment, multiple services, numerous users and applications. Also, each component must present some of the different functionalities previously stated. With respect to the intended functionalities, not all components need to be specified. Also, the level of details to be considered for a given component may vary. This phase is intended to help stating the intended functionalities of the application to be designed and focusing on the relevant aspects only.

The application to be specified may concern an existing domain. For instance, network management applications are usually defined for existing networks. Therefore, the application domain already exists and the new functions must cope, at the appropriate abstraction level, with existing components. Generally, the design of an application starts with a (possibly empty) given domain, and new components or new aspects of existing components are added. Such an approach promotes re-use of specifications since it does not isolate a given application from existing and future ones. Also, this promotes some "orthogonality" principles since different functional aspects can be added without modifying existing ones.

The last point to consider is the intended purposes of the specifications. We use the term system model to denote the specification of a domain together with its functionalities. Such a model may be used for various purposes:

(1) Formal verification, to check the consistency of the design, and/or the correctness of algorithms.

(2) Documentation for some hardware and/or software architecture.

(3) Simulation, for user training, for future system development analysis, or as a prototype before building a larger scale system.

(4) Performance analysis.

(5) Automated software production.

Different formalisms may be required to achieve these different goals. Even with the same formalism, different specification styles may be adopted. For instance, "intentional" or "extensional" styles might be preferred (see Section 3.6). Also, the level of details to be included in the model greatly depends on its intended use.

The results of this preliminary phase are more of a set of guidelines for the following phases than formal results. They should help the designer to focus on relevant purposes and to determine the level of abstraction required for each component.

As an example of such a result, see the informal, textual, description of the database example in Section 4.1. Only the structure, cost, and weights of the parts are considered though it is very likely that a real manufacturing database would include many other aspects.

Though this phase is presented as a preliminary phase, our experience showed that these issues should be constantly considered throughout the entire design process.

2.2 Phase 2: Domain Definition

The domain definition phase is intended to capture the relevant elements of the existing or foreseen domain, together with their essential characteristics. The result should be a description of the domain as a set of entities together with the various relationships among them that are relevant to the functionalities and purpose of the model. This phase can be refined in the following sub-phases:

(1) The first one is to identify entities of interest, together with their specific characteristics. Various entities exist, and are usually characterized with specific properties covering different aspects such as state, value, and structure. These entities can be represented as objects and attributes whatever the target language is.

(2) A second sub-phase is the identification of the various relationships existing among these entities. They usually cover different aspects of the application:

(2.1) Structuring aspect, represented by the aggregation relationship, often stated as "is-part-of" or "is-made-of" relationships; this relationship helps to define appropriate abstraction levels; i.e., whether a given entity must be handled as a whole or as a set of components; aggregation can be static or dynamic.

(2.2) Typing aspect, represented by the inheritance relationship, often stated as "is-a" relationship; different entities in the domain may share some common characteristics which can be specified in a generic template (or type); templates can be specialized for various purposes; e.g., to specialize inherited features or to add some new ones.

(2.3) Functional aspects: many identified relationships stem from the functionalities the designer intends to specify; also, some of these relationships are static while some are dynamic. Object-oriented languages allow a designer to formally specify these relationships: aggregation and functional relationships are represented by means of attributes, while subtyping is well captured by the inheritance mechanism.

Examples of entities belonging to the domain described in Section 4.2 are the "Part", "BasePart", "CompositePart", and "Supplier" entities. Aggregation is illustrated by the "CompositePart" entity made of "CompositePart" or"BasePart" entities. Inheritance is illustrated by "BasePart" and "CompositePart" entities both being "Part". A specific "Supplied" relationship exists among the "BasePart" and "Supplier" entities.

(3) The next sub-phase is consistency checking, which applies to both entities and relationships:

(3.1) Entities having common attributes might be specialized instances of more generic types; common characteristics may be grouped within common templates.

(3.2) An entity may include orthogonal (unrelated, or independent) aspects; therefore, it may be defined as inheriting of some more general entities separately specified.

(3.3) Relationships among entities may be represented as attributes of these entities; some relationships may be better represented as specific entities.

(3.4) Relationships may lead to define entities functionalities; i.e., they may be defined as operations, rather than attributes, during the next phase; (see Section 2.3).

(3.5) Relationships must be considered with respect to the inheritance lattice; a relationship stated for general templates might not hold as such when coming to more specialized entities, and thus should be more precisely defined.

(3.6) Some relationships may have constraints; as an example of such a constraint, see the "Acyclicity" applying to the structure of the database example in Section 4.2.

(4) A last sub-phase is to (re)write a design documentation where the identified components appear clearly. Since the formal model may be less readable , the textual, informal, documentation should closely match the system model.

2.3 Phase 3: Functions Definition

The function allocation phase intends to distribute the required functionalities among the identified entities. There are four important aspects to this process:

(1) Since entities are represented as objects, the functions they have to perform are defined as operations they must offer. An operation is formally defined as a procedure or a function; i.e., with some input and output parameters, which must be objects too. These parameters must be defined if possible.

(2) Having identified the operations, the designer must allocate them to some objects. For each operation, the designer must consider which entity seems the "most natural" one to offer the operation. Several points are to be considered:

(2.1) When an operation can be offered by several entities, it might be better to allocate it to a common ancestor in the inheritance lattice.

(2.2) When an operation does not seem to fit a particular entity, or seems to naturally belong to very different entities, or seems to correspond to some cooperative processing by several entities, it might be convenient to allocate it to a new "support" entity introduced for the purpose of allocating the operation.

(2.3) Operation parameter types should be specified with respect to the entities offering them; for instance an operation offered by a high-level entity might have some general parameter types, but these parameter types will be refined when the operation is inherited.

Examples of operation definition and allocation to entities are shown in Section 4.3. The "cost" operation is allocated to the "Part" entity. The result of this operation is an integer.

(3) The next point is to consider the support entities introduced during the allocation process and to integrate them within the application domain. The same process as in Phase 2 should be reapplied; this favors the re-use of software specifications since the resulting structured domain can be re-used when defining future applications in the same field. In Section 4.3, the "DBInterface" support entity is introduced to offer the "compute_cost" operation.

(4) Since this phase may lead to new entities and operations they offer, the textual specification should be rewritten such that all identified entities appear clearly and their interactions are expressed in terms of operations.

2.4 Phase 4: Behaviors Definition

This last phase consists of the definition of the behaviors of the various entities for the allocated operations. This process mainly depends on the knowledge and expertise of the designer in the specific field. However, some general principles can be applied:

(1) For each object, it is first necessary to define the accepted sequences of operations.

(2) For each operation offered by a given object, there are two possibilities:

(2.1) The behavior for that operation is simple enough so it can be easily specified with the language statements; the necessary processing is described in terms of state changes, attribute modifications, and interactions with other objects.

(2.2) The behavior is complex, and a refinement process can be applied to it; the technique is to consider the processing to be performed as a specialized "application" which can be specified by repeating the design process from Phase 2 to 4, until all components are fully specified.

As an example of behavior specification without refinement, see the algorithms for the "compute_cost" and "cost" operations in Section 4.4.

3. Characteristics of Mondel

As mentioned above, the design of Mondel was influenced by many different existing languages. This section provides a discussion of the considerations that lead to the particular choices made for Mondel. Table 1 provides an overview of the features of various programming and specification languages. Most features indicated in the table are discussed in more detail below. None of these existing languages included all those features that we considered important for the kind of applications foreseen.

3.1. Standard Features of Object-Oriented Languages

Mondel adopts many language elements which are common in most object-oriented languages, such as the following.

(1) **Object types, instances and classes**: A specification defines a certain number of object types. Each newly created object instance belongs to a given type which defines its properties. Using the inheritance structure discussed below, a given type definition may be used to define more specific subtypes of objects. We say that an object instance belongs to the object **class** *A* if it either belongs to the type *A* or one of the subtypes of *A*.

(2) **Attributes**: An object type definition may include a certain number of named attributes. This means that each object instance of that type will have fixed references to other object instances, one for each attribute. A typical example is an Employee object which has attributes named First_name, Family_name, Sex and Salary. An attribute may be declared non-visible; by default, an attribute is visible which means that any object "knowing" the object may also access its attribute. It may also be declared internal, which means that it is defined by the internal behavior of the object; otherwise it must be provided as effective parameter when the object instance is created. For instance, the attribute Salary could be internal and its value could be determined by an operation Fix_salary. The other attributes of an Employee could be provided when an Employee instance is created.

		COMMUNICATION MECHANISM	EXCEPTION	FORMAL SEMANTICS	GENERICITY	INHERITANCE	OBJECT SUPPORT	PARALLELISM	PERSISTENCE	READABILITY	TYPING
Ada	[Bray 85]	PC	X		X		Ltd	X		High	X
Argus	[Lisk 88]	PC			?	?	X	Intra-object	X	Average	X
C++	[Lipp 89]	PC				X	X			Average	X
Eiffel	[Meye 88]	PC	X		X	X	X		Ltd	High	X
Emerald	[Blac 87]	PC with queue			X	X	X	Ltd		Average	X
Lotos	[Iso 87c]	Multi-Rv		X	Ltd	X		X		Low	X
Modula 3	[Card 88b]	Co-routines	X		X	X	X	Co-routines		High	X
Mondel		Multi-Rv, PC	X	X	X	X	X	Intra-object Inter-object	X	High	X
Pool	[Amer 87]	PC with queue		X	?	?	X	X		High	X
Smalltalk	[Gold 83]			Ltd		X	X	Ltd		Average	
Traces	[Hoff 88]			X			X	X		Good	

PC: Procedure Call
Rv: Rendezvous
Ltd: Limited

Table 1 Comparaison Chart

(3) **Operations**: The definition of a type may include the declaration of named operations (sometimes called "methods") which may be invoked by other objects on object instances of that class. Each operation has a fixed number of parameters (which are objects), and it may return an object as result.

(4) **Typing**: Mondel supports strong type checking based on the declared object types. Generic classes (i.e. with type parameters) are also supported. Therefore the type consistency of the effective parameters of operation invocations and object instantiations can be checked by a compiler, except in certain cases involving generic types. The language also includes an operator to dynamically determine the type conformance relation between type parameters. A CASE construct allows performing different actions depending on the type of a given parameter or the result of an operation.

(5) **Behavior**: The execution of an operation implies in general certain state changes and possibly the execution of other operations on other objects which are "known" to the object which executes the operation. We call "behavior of an object type" the information which is known about the execution of operations and the initialization of a newly created object instance.

3.2. Methodological Aspects: "Everything is an Object"

Simplicity is important for language design. The approach that "everything is an object" [Amer 87, Blac 87] limits the number of concepts to be introduced. This is in contrast to LOTOS [ISO 88b] which includes "processes" and "abstract data types", or Argus [Lisk 88] which includes normal objects and "guardians" which have special properties.

However, it is important to verify that important concepts found in the systems to be specified can be naturally modelled with the concepts that the language provides. The following comments are intended to show how certain important system aspects can be described.

(1) **Record structure**: The familiar "record" data structure can be modelled by an object type which has attributes that correspond to the fields contained in the record structure.

(2) **Entity/Relationship models** (see also Section 2): Mondel distinguishes the class of persistent objects which must explicitely be deleted; normal objects are considered to disappear as soon as no reference to them exist from any persistent object instance (automatic garbage collection is assumed). Entities and instances of relation tuples can be modelled by persistent objects. The entity attributes are simply modelled by

normal Mondel attributes, however, relationships may be modelled in different manners.

(3) **Union types**: Variant record structures are found in most programming languages; the fact that an object class includes several different subclasses can be expressed in the inheritance structure of the types definitions (see below). Mondel includes a CHOICE construct which allows the straightforward declaration of such an object class. The CASE construct mentioned in Section 3.1 can be used to determine to which subtype a given object instance belongs. If type parameters are considered as attributes (see Section 5.4), then the CHOICE construct may also be used to define enumeration types, such as: **type** colour = **choice** blue, green, red **endtype.**

(4) **Variables**: While many aspects of object-orientation can be combined with the functional programming paradigm, there are often aspects of systems that are naturally modelled by the changing state of an object. While the attributes of an object refer to a fixed object acquaintance (which is determined during the initialization of the attribute), it is often convenient to use attributes of variable type. For this purpose, Mondel includes predefined generic object classes, such as VAR, SET, SEQUENCE which may be used to define attributes which may change their state. For instance, an attribute of class VAR[Integer] is an integer variable which may be updated through the predefined operator ":=" in the usual manner, and be read through the operator "^". SET and SEQUENCE objects represent typed sets or linear lists of objects, respectively. Like VAR objects, their "content" may be read and changed through the usual predefined operations.

It is noted that the ASN.1 notation [ISO 87b] used for the description of open distributed systems standards has been considered during the design of Mondel. In fact, the concepts found in that language can be naturally written in Mondel [Boch 90b].

3.3. Communication and Concurrency

Three approaches to introducing concurrency in object-oriented systems have been considered [Amer 87]:

(1) The objects of the systems are activated by several concurrent "processes".

(2) Operation calls are asynchronous, that is, an object calling the operation of another object may continue its activities in parallel with the object executing the called operation.

(3) Each object is a "process" (approach taken in Pool [Amer 87]).

In Mondel, we generalize approach (3) by allowing for parallel activities within a single object instance [Trip 89]. The behavior of an object can be defined by a PARALLEL construct which includes independently parallel activities, similar to "behavior expressions" in LOTOS [Loto 87c]. In addition, Mondel includes a FORALL construct which allows the parallel execution of a certain task for all object instances of a certain class, which is similar to ESTELLE [ISO 88a] and Argus (FOREACH construct, [Lisk 88]).

In this context, the issues of inter-object communication must be discussed. Three kinds of communication primitives may be distinguished:

(1) Message passing where the called party is not synchronized with the caller (e.g. as used in Actor systems [Agha 87]).

(2) (Remote) procedure call (RPC) where the caller will wait until the callee has issued (possibly implicitely) a RETURN statement (used in many languages, such as Pool, Argus, and ADA).

(3) Full rendez-vous interaction, where the caller obtains implicitely feedback about the willingness of the caller to execute the requested operation, and may persue other alternatives as long as the callee has not started the execution. Certain languages allow rendezvous interactions with more than two processes [Buck 83, Loto 89], and the use of parameter values to determine which one of several processes will be involved in a given rendezvous.

We have chosen synchronized communication because it is more suitable for the description of abstract interfaces [Boch 90d]. According to the RPC paradigm, it is assumed that the calling object "knows" the called object on which the operation is invoked (Note: This is in general not required in the case of "full rendezvous"). It is clear

that the RPC paradigm is easier to implement in a distributed environment (see for instance [Blac 87]) than full rendezvous (see for instance [Gao 89]).

In most cases, Mondel specifications use the RPC approach. However, the use of the CHOICE statement allows the calling object to foresee alternate actions which may depend on the readiness of the called objects and certain restrictions on parameter values which the latter may impose in the form of PROVIDED clauses associated with the acceptance of an operation call.

3.4. Conformance Relation and Inheritance

Various forms of inheritance have been considered in different object-oriented languages (e.g. [Blah 88], [Card 88a]). While often only simple inheritance is considered, that is, each type has at most one parent (supertype), Mondel allows for a form of multiple inheritance where a given type may inherit from several supertypes, as long as the inherited properties are without contradiction.

The intention is that an instance of a subtype can be used in any specification context where an instance of one of its supertypes can be used. We call this relation in the following "conformance": a subtype conforms to the more general supertype. There are different aspects of object behavior that are relevant to the conformance relation, such as the following (see also Boch 89]).

(1) **Sets of values:** The set of objects belonging to a subtype is included in the set of objects belonging to its supertype.

(2) **Attributes:** A conforming object has (at least) all the attributes defined for the more general object type. The attributes may be more specialized (conforming).

(3) **Operation signatures:** A conforming object has (at least) all the operations defined for the more general object type, where the operation result must be conforming and the input parameters must be inversely conforming (see for instance [Blac 87]).

(4) **Behavior of operations:** The effects of the operations of the refined type satisfy the requirements specified for the more general object type. This includes the

requirements for the obtained results, as well as the resulting state changes induced on the object executing the operation.

(5) **Blocking properties:** If the supertype object blocks the execution of a given operation in certain situations, then a refined object must also block in such situations.

The conformance relation has an important role to play for the validation of specifications in the form of type checking. As most languages that include strong type checking, Mondel requires that each effective parameter of an operation call (or each effective attribute of a newly created object) is an object which conforms to the class of the corresponding formal parameter declaration (or attribute definition). Most of these checks can be done at compilation time.

For the aspects (2) and (3) above, it is possible to define the conformance relation between object types based on their structure [Blac 87]. In contrast, in Mondel the conformance relation is based on the inheritance relation among object types as explicitely included in the text of the specification. The compiler can check that these conformance requirements are met when the inheritance relation is used. The other conformance requirements, however, are difficult to check automatically and should be verified by the person writing the specification.

The advantages and disadvantages of inheritance with type checking and generic types (i.e. types with formal type parameters) has been discussed in the context of Eiffel [Meye 86]. We follow this approach by allowing for user defined generic object types. Although Mondel introduces specific syntactic constructs for the type parameters in order to make the use of generic classes more readable, we think that, formally, the effective type parameters could be considered as instances of the class TYPE, and a type parameter as an attribute of class TYPE, as discussed in Section 5.4.

3.5. State Transitions

In the design of functional languages, one tries to avoid the concepts of "state" and "state transitions". However, in many applications, in particular for distributed systems, the concept of the state of an object seems important, since the result of an operation may depend on the past history in which the object performing the operation was involved. As

far as its future behavior is concerned, the past history of an object instance is represented by its state. As in Emerald [Blac 87], Mondel distinguishes objects that never change their state, such as integers. In general, operations that do not change the state of an object are called "pure" (as in ESTELLE [ISO 88a]) and the calling syntax for pure operations is distinguished to make specifications more readable (The symbol "." is used to introduce the call of a pure operation, while the symbol "!" is used for other operation calls).

The state of an object instance is determined by the place in the behavior definition where the point of execution control of the object resides. In addition, the states of the attributes have an influence on the behavior of the object. The attributes may in particular be of the predefined types VAR, SET, SEQUENCE, the content of which may be changed through the execution of the corresponding predefined operations, such as assignment ":=" or "add element".

While the state changes of the different objects throughout the system may proceed in parallel, unless they are explicitely synchronized through operation calls, Mondel also includes the concept of transactions which are atomic operations, as known from the database area [Gray 81]. The execution of different atomic operations is assumed to be scheduled such that their execution appears to be sequential (serializable). Like most proposals for nested atomic actions (e.g. Argus [Lisk 87, Lisk 88] and Locus [Muel 83]), Mondel has automatic recovery after failures only at the highest level, not for subactions.

3.6. Specification Styles

A well-know method of software specification takes an approach where two complementary languages are used: (1) The program is specified by assertions, usually in the form of input/output assertions associated with the procedures or functions, and invariants for program loops and shared monitors. (2) The program is implemented by an algorithmic description in the form of program code. Program verification, then consists of verifying that the program code satisfies the assertions and invariants if the environment of the program satisfies the input assertions.

Two descriptive approaches, assertional and algorithmic, can also be found in the specification language LOTOS [ISO 88b], where the behavior expressions (based on a modified CCS [Miln 80] are largely algorithmic, while the data type definitions,

following an algebraic abstract data type approach [Ehri 85], are largly assertional. It is interesting to note that the latter specifications become "executable" by using a rewriting formalism, provided certain restrictions on the form of the specifications are observed. Both approaches can often be used to describe the same given behavior [Gotz 86].

It is clear that a specification language can be used for writing specifications in different styles. Sometimes, different specifications of the same system are developed where one specification is more defined or more detailed than another. Sometimes they are assumed to be equivalent. The algorithmic style is often closer related to implementation, while an assertional style leads often to more abstract specifications. Different styles related to the LOTOS language are discussed in [Viss 88]. The so-called "constraint-oriented" style is mentioned in particular which allows the separate specification of different constraints that must be satisfied by the object behavior. Multiple inheritance can be used to obtain this specification style in many cases. However, in certain cases, the different constraints are embodied by different objects. To describe such situations, Mondel introduces the concept of a "controlled" attribute which is an object onto which the controlling object imposes its own constraints. This has been used to describe the abstract interfaces contained in the OSI reference model [Mond 90b].

In Mondel, the algorithmic style is supported relatively directly by the statements of the languages. The assertive style is also supported, in particular through the following features:

(1) **Assertions**: An ASSERT statement may be included anywhere in the behavior definition of an object. It verifies that a specified assertion is satisfied. Furthermore, in the absense of an algorithmic behavior definition, the ASSURE statement indicates which output assertions will be satisfied by the specified operation or procedure.

(2) **Invariants**: Invariants may be defined for each given object type or may be associated with a group of definitions. The invariants define assertions that will be validated after the execution of each transaction operation (see above). This construct may be used to describe integrity constraints for the defined data structures.

(3) **Assertive behavior definition**: As an alternative to the algorithmic specification style supported by the Mondel statements, the use of an assertive style has been considered, but not yet elaborated. Such a style may be more appropriate for certain

types of object behaviors. For instance, the behavior of the predefined object types have been specified using the "Traces" notation of [Hoff 88].

4. An Example

In this section, we illustrate the language Mondel and the object-oriented design methodology with a simple database example taken from [Atki 87]. The Mondel specification is given in the Appendix. In the following sections, we discuss how this specification has been obtained following the phases of the methodology described in Section 2.

4.1 Phase 1: The Problem

The database represents the inventory of a manufacturing company. It consists of a set of parts in which we distinguish the composite and base parts. The former corresponds to the parts manufactured by the company whereas the latter corresponds to those bought from the outside. The database represents, in particular, the way the composite parts are manufactured; i.e. the subparts involved (which may themselves be composite), the cost of such an assembly and the mass increment. Note that an instance of part represents a particular kind of part and its characteristics; the database does not record the individual parts processed by the factory. It makes sense that, for example, part A is used in the manufacture of parts B and C which are both used in the assembly of part D. The composition relationship corresponds therefore to a directed acyclic graph. Finally, for the base parts, the database must record the purchase cost, the total mass and other relevant information concerning the different suppliers.

Our task consists of formalizing the database description and defining three simple transactions as follows:

Task 1: Describe the database.
Task 2: Select all the imported parts that cost more than $100.
Task 3: Compute the cost of a composite part.
Task 4: Record a new manufacturing step in the database, that is, how a new composite part is manufactured from subparts.

4.2. Phase 2: Description of the Application Domain

This phase yields an enumeration of the relevant entities and relationships. It consists of two results:

(1) An entity/relationship diagram, which offers a qualitative description of the domain (see Figure 1).

(2) A partial Mondel specification.

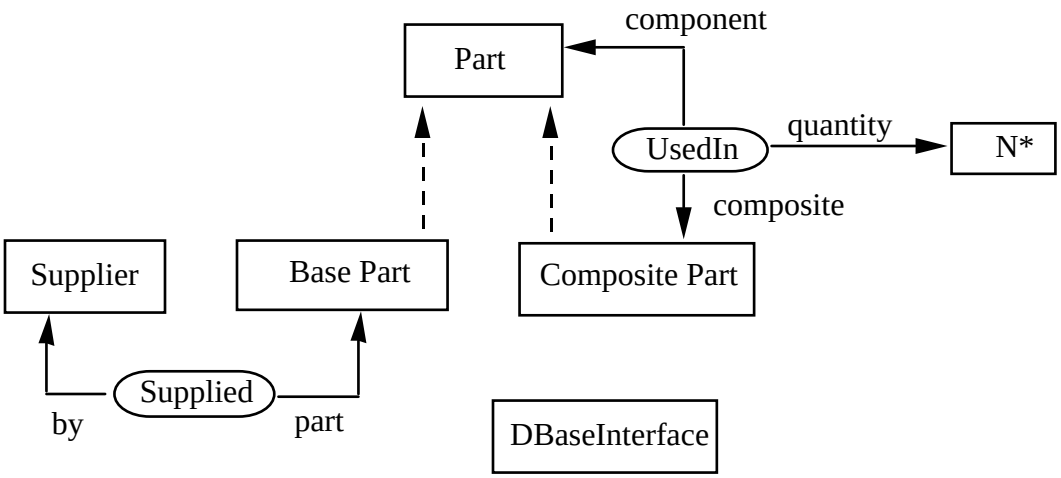

Figure 1. Entity/Relationship Diagram

The entities are named Part, BasePart, CompositePart and Supplier. They are shown as boxes (we do not consider the DBInterface yet; it will be discussed in Section 4.3). The relationships are named Supplied and UsedIn and are shown as ovals. N* denotes the set of positive integers and is used to quantify the UsedIn relationship. Every entity or relationship is represented in Mondel as a type definition.

The Part type represents all parts involved in the manufacturing process. The CompositePart and BasePart types are two refinements of Part. BasePart represents the imported parts whereas CompositePart represents those manufactured within the company.

The relationship Supplied links base parts and their suppliers. The UsedIn relationship defines the decomposition of composite parts into components. For example, let "u" be an instance of UsedIn. One can say: "u.quantity of u.component(s) are used in the manufacture of u.composite".

4.2.2 The Attributes

In contrast to the diagram of Figure 1, the Mondel specification in the Appendix includes also the attribute definitions of the entities and the relationships (as well as the operation and behavior definitions discussed below). For instance, consider the following partial specification:

```
type Part = Entity with
        name  :  String  key;  ...

type BasePart = Part with
        purchase_cost   :  Integer;
        total_mass      :  Integer;  ...

type CompositePart = Part with
        assembly_cost     :  Integer;
        mass_increment  :  Integer;  ...
```

It shows the way the attributes are defined in Mondel. The Part type has an attribute "name". As indicated by the keyword KEY, each instance of Part is uniquely identified by the value of this attribute. In addition to the inherited attribute "name", the BasePart type has the attributes "purchase_cost" and "total_mass". Similarly, the CompositePart type has two more attributes defining the assembly cost and the mass increment resulting from the assembly.

Attributes are also used to express the plain arrows of the Figure 1. As an example, we have the following relationship definitions:

```
type Supplied = RelationShip with
        part   :  BasePart    key;
        by     :  Supplier    key;
endtype Supplied
```

```
type UsedIn = RelationShip with
        composite   : CompositePart   key;
        component   : Part            key;
        quantity    : Integer;
endtype UsedIn
```

The first two attributes of UsedIn and Supplied indicate the entities related by an instance of these relationships. In addition, UsedIn has an attribute (in the E/R model sense) named "quantity". The keywords KEY indicate that an instance of Supplied or UsedIn is uniquely identified by the combination of two attributes.

4.2.3 Acyclicity

According to the requirements given in Phase 1, the <composite, component> relationship must be acyclic. In order to formally describe this constraint, we introduce a pure boolean operation "use" on the Part type. An expression such as p.use(p') yields true if and only if there exists a non-empty path in the <composite,component> graph starting from p and ending at p'. The acyclicity property of UsedIn is specified as the following invariant:

```
invariant
        forall  p : Part  do
                assert  not  p.use(p);
        end;
```

That is, a part p cannot be a used as a subpart of itself.

4.3 Phase 3: Identification of Operations

Operations must be introduced for handling the Tasks 2 to 4 listed in phase 1. For Task 2, one can imagine an operation named "select_base" with a parameter "threshold" and defined by the following algorithm:

```
procedure select_base(threshold : Integer) : Set[String]  =
        define result = new Set[String]   --   Defines a new set initially empty.
        in
                forall part : BasePart
                suchthat  part.cost >= threshold do
                        result!put(part.name);      --  Collects the Part names.
                end;
        end;
        return result;  --  Transmits the result to the caller.
endproc select_base
```

The question is: Which objects should be attributed with this operation? This task concerns only the BasePart entities. Therefore, defining "select_base" as a BasePart operation seems appropriate. However, in the case where no BasePart instance exists, the "select_base" operation cannot be invoked, even if it makes sense. Consequently, we defined the DBInterface type to support the "select_base" operation. It is also explicitly shown in the E/R diagram to emphasize the fact that it is an entity part of the specification. Similarly, we defined a "compute_cost" operation for Task 3.

```
type DBInterface = Entity with
operation
    select_base     (threshold  : Integer) : Set[String]    pure atomic;
    compute_cost    (part_name : String)  : Integer         pure atomic; ...
```

Only the operation signatures are provided, that is, the operation names, the parameter names and types, and the result types. Other qualifiers are also given. For example, a pure operation has no side effects. Atomic operations correspond to transactions (see section 3.5). We do not consider the details of Task 4. A signature might be attributed to the interface in the same way as above.

4.4 Phase 4: Behavior Definitions

In this phase, we make the operation definitions more specific. The algorithms proposed in the Appendix use standard recursive techniques. For example, let us consider the "compute_cost" operation. Its algorithmic definition is given by a procedure with the same name defined in the WHERE clause of the DBInterface type.

```
procedure compute_cost(part_name : String) : Integer =
        ifexist  p:Part
        suchthat  p.name =  part_name then  return  p.cost;
        else   ...
```

The IFEXIST statement selects a part with the specified name and return its cost. This implies support of a "cost" operation by all part instances. We define the following signature:

```
type Part = Entity with ...
operation
        cost   :  Integer  pure; ...
```

However, the algorithmic definition is not provided in the Part type. It depends on whether the part is a base part or composite part. In the case of a base part, the cost is simply given by the value of the "purchase_cost" attribute, as defined by the "cost" procedure in the WHERE clause of the BasePart type. For a composite part, it corresponds to the value of its "assembly cost" attribute added to the costs of all its subparts. This recursive definition appears in the WHERE clause of the CompositePart type (see the Appendix).

The definition of Task 4 is not given. It can be easily constructed based on the following elements. Object creation is achieved using the Mondel operator "new". For example, the expression "new BasePart(100, 10) as Part("foo")" creates a new BasePart part named "foo" with purchase cost 100 and total mass 10. Care should be taken that a sequence of such expressions within the scope of a transaction preserves the constraints expressed in the specification (e.g. invariant, key).

Finally, a behavior must be provided to define the sequence in which the operations may be invoked. The keyword "behavior exclusive" is a shorthand notation to express the fact that the operations are executed sequentially. Other sophisticated behavior descriptions are possible but not needed here.

5. Verification/Simulation/Implementation Issues

The full definition of a programming/specification language consists of two parts: i) the syntax and static semantics, and ii) the dynamic semantics. The syntax and static semantics define how basic symbols can be combined to build valid language sentences. For Mondel, this part has been defined using context-free and attribute grammars [Leco 89] and has been implemented by a compiler written in Prolog, which also generates the internal representation used by the Mondel interpreter described below.

The dynamic semantics associates a meaning to the valid language sentences. The dynamic semantics can be defined using complementary informal and formal methods. Informal methods use natural language such as English which is a notation that can be understood by a large group of people. Nevertheless, this type of semantic description is often ambiguous and open to many different interpretations. On the other hand, formal methods use mathematical notations. A precise language description can be obtained and the implementation is therefore easier to make. Moreover, we get the possibility of formal verification (to be discussed in Section 5.3).

To define the formal semantics of Mondel [Barb 90a] we adopted the operational approach. In this approach an abstract machine executes a specification. The meaning of specification statements is expressed in terms of actions made by the abstract machine. We more particularly applied the technique of Plotkin [Plot 81] where state/transition systems are taken as machine models. This technique has also been applied to define the semantics of the POOL object-oriented language [Amer 86]. In our approach we use a slightly different notation. We use first order logic restricted to Horn clauses [Kowa 79] to define the inference rules. The translation of the dynamic semantics into Prolog is straightforward and we were able to obtain a simulation environment in a relatively short time period [Will 90]. The simulator is described in Section 5.2. Finaly, Section 5.4 discusses a new version of Mondel being developed which supports the concept of dynamically modifiable specifications.

5.1 Formal Definition of the Dynamic Semantics

The execution of a Mondel specification is modelled by a set of states with transitions between them, starting from a specific initial state. The transitions are given by a transition relation denoted "$\rightarrow$" (arrow) with the functionality:

$$\rightarrow \subseteq \text{States} \times \text{Actions} \times \text{States}$$

Where *States* is a set of states and *Actions* a set of actions. The transition relation is defined inductively by a set of inference rules. The dynamic semantics of Mondel has therefore the two aspects of representation of states and definition of inference rules. We first discuss state representation.

A state is a set of active objects. We have an abstract representation for objects. Let us consider the following Mondel type definition:

```
type A = object
        with x:B;
behavior
        new C(x);
endtype
```

Instances of type *A* have an attribute *x* of type *B*. Their behavior consists of the single statement *new* which creates an instance of type *C*. An active object of type *A* will be represented as the following tuple:

<id1, A, { id2/x }, new C(x)>

The first component of the tuple is a unique identifier generated for this object. The second element is the object type name. The third component is a binding which associates actual objects to formal attributes. In this particular example the object reference id2 is associated to the formal attribute *x*. The last component represents the statement being executed by the object. When *new C(x)* is executed the state is expanded with a new element of type *C*.

In general the semantics of an active object is obtained by combining together several inference rules. For the sake of legibility, inference rules are structured into different levels. The lowest level is the so-called "object level". As an example, we show the rules related to object creation. At the object level we have the following rule:

if

Obj = <Id, T, Bind, new TypeName(Id1,...,Idn> (1)

fattr(TypeName) = AttrName1,...,AttrNamen (2)

Bind ={ Id1/AttrName1,...,Idn/AttrNamen } (3)

NewId = newsym (4)

Obj' = <Id, T, Bind, NewId> (5)

then

$\rightarrow$(Obj, new(TypeName,Bind,NewId), Obj') (6)

This rule says, on line (6), that the object *Obj* can execute the action *new(TypeName,Bind,NewId)* and be transformed into *Obj'* if the preconditions of lines (1) to (5) are fulfilled. Line (1) defines the structure, i.e. the state, that *Obj* must have. On line (2), the function *fattr* yields for every type name its ordered list of formal attributes. This list is used on line (3) to define the binding of the created object for which a new identifier is generated on line (4) by the *newsym* function. The *newsym* function is not in pure first order logic and is introduced for the sake of simplicity. It can easily be translated into pure logic by keeping track, from one state to another, of identifiers in use. At last, line (5) defines the successor state *Obj'* of the creator.

The full definition of object creation also requires a higher level inference rule defining the transition from a set objects to a new set of objects which will contain the new instance:

if

S = A + { Obj } (1)

$\rightarrow$(Obj,new(TypeName,Bind,NewId),Obj') (2)

S' = A + { Obj', <NewId, TypeName, Bind, getinitbeh(TypeName)> (3)

then

$\rightarrow$(S,new(TypeName,Bind,NewId),S') (4)

This rule defines, on line (4), a state change of the set of objects *S* on action *new(TypeName,Bind,NewId)* with successor state *S'*. Line (1) describes the structure of *S* which is a set of objects *A* plus an object *Obj*. On line (2) we say that *Obj* is the creator of the new object. Here we apply the object level inference rule defined above. Line (3) defines the structure of the successor object set. The function *getinitbeh* yields for every type name the corresponding initial behavior.

5.2 Prolog Interpreter

Implementing a language while it's syntax, static and dynamic semantics are unstable requires a host language suitable for rapid prototyping. With Prolog, it was possible, during the development of Mondel, to have a working implementation that could keep up with the periodic language updates.

The syntax analysis and the static semantics checking of Mondel specifications are made by a compiler which generates a collection of Prolog terms. This intermediate representation is equivalent to the Mondel text, except that some compiling information is extracted and the code is flattened such that every identifier becomes globally unique.

The translation to Prolog terms is quite straightforward. For example the Mondel type *A*, introduced in Section 5.1, would be represented as follows:

```
type(t3, [[attr(V1,a1),t2]], new(t1,[[attr(V1,a1),a3]]).
ren(t1,'C').
ren(t2,'B').
ren(t3,'A').
ren(a1,'x').
ren(a3,'y').
```

where *ren* (rename) facts are used to bind internal names with names appearing in the source text. Here it is assumed that the type *C* has a formal attribute named *y*. A term *attr* pairs a Prolog variable with an attribute, to store the actual value.

The example below shows the object creation rule of Section 5.1 translated into Prolog. It can be understood as follows: A low level transition *new(..)* is possible on object OBJ that results in the modified object OBJP.

```
low_trans(OBJ, new(TypeName, Bindp, NewId), OBJP) :-
        % This object is ready to execute the NEW statement
                OBJ = obj(Id, Type, Bind, new(TypeName, ActualAttr)),
        % Get the formal attribute list of type TypeName
                fattr(TypeName,FormalAttr),
        % Bind actual attribute values to formal attributes
```

```
        bind(ActualAttr,FormalAttr,Bindp),
    % Generate a unique identifier for the new object
        newsym(NewId),
    % Next state of object OBJ
        OBJP = obj(Id, Type, Bind, NewId).
```

Notice that this rule does not create a new object. This is done by the following higher level rule which refers to the rule above.

```
high_trans(STATE,new(TypeName, Bindp, NewId),NEXT_STATE) :-
    % There is one object OBJ from STATE
        partition(OBJ,STATE,REST_STATE),
    % That can perform a low level transition "New"
        low_trans(OBJ,new(TypeName, Bindp, NewId),OBJP),
    % Get type and use it as a template
        type(TypeName,Bindp, Behavior),
    % New created object
        NEW_OBJECT = obj(NewId, TypeName, Bindp, Behavior),
    % Build the next system state
        NEXT_STATE = [OBJP, NEW_OBJECT | REST_STATE].
```

The heart of the interpreter is a set of inference rules, similar to above, which we obtained from the formal semantics of Mondel by translation into Prolog. In addition, the *sys* predicate, defined below, plays the role of an inference engine. It takes as input a set of Mondel objects (STATE) and tries to apply an inference rule. The resulting state (NEXT_STATE) becomes the input to the predicate in the next step. The process will stop when no more transitions are enabled, that is a deadlock has occurred.

```
sys(STATE,TRANSITION,NEXT_STATE) :-
    % Try a transition
        high_trans(STATE, TRANSITION, NEXT_STATE),!,
    % NEW_STATE becomes the new system state
        sys(NEXT_STATE,_,_).
```

The similarity between the formal semantics rules and the Prolog rules is obvious and shows that Prolog is well suited for fast implementation. However, a major drawback of

Prolog is its slow execution and its hunger for computer space. This can be alleviated by reducing data that is passed between rules. For instance, one could store objects as dynamic Prolog clauses by using Prolog *assert* and *retract* predicates; then the objects state will not be passed from one rule to another as the "$\rightarrow$" (arrow) relation is applied.

5.3 Verification

The goal of verification is to determine if a system specification is correct and satisfies certain properties. Properties can be classified into two categories: (1) general properties, and (2) semantic properties. General properties are independent of the system function. Examples are finiteness, termination and deadlock-freeness.

On the other hand, semantic properties are closely related to the system function. For example, in the field of protocols, conformance of the protocol to the service is a semantic property always required. In our approach to the verification of Mondel specifications, we use assertions on states to define the semantic properties.

The verification of properties of Mondel specifications can be done using two approaches: (1) a simulation based approach, and (2) an analytic Coloured Petri Net (CPN) based approach. With the help of the Prolog simulator described above, specifications can be executed to discover bugs. This method covers the full Mondel language. However, in the general case, the analytical approach is partial because not all possible execution paths are explored.

We developed a second verification approach [Barb 90b] which works for an interesting subset of Mondel and uses the theory of CPNs [Jens 87]. For this Mondel subset, a CPN operational semantics has been defined. This semantics is consistent with the semantics discussed in Section 4.1 and permits direct application of the CPN reachability analysis technique. This reachability analysis method is particularly interesting because it can reduce the explored state space by exploiting symmetry relations between states.

Similar Petri net based approaches have been developed for languages with anonymous processes such as Lotos [ISO 88a]. Processes are modelled as unstructured Petri net tokens. In opposition to these anonymous processes, a Mondel object has an identity and a set of acquaintances; i.e., identifiers of other objects. We discovered that these aspects

can be modelled quite nicely by CPNs. Objects are modelled as structured tokens which represent identifiers and acquaintances.

Several authors have proposed the application of Petri net verification methods to high-level language specifications. Various approaches described for LOTOS [Gara 89, Marc 89] and ADA [Mura 89, Shat 88, Suzu 90] involve the translation of the specification written in these languages into Petri nets. In contrast, we have shown that an important Petri net verification methods can be adapted to be directly applied to a subset of the LOTOS language [Barb 90b]. Similarly, the approach described in this paper.

By virtue of the CPN semantics for Mondel, we apply the reachability analysis technique for CPNs [Jens 87] directly to Mondel. A set of active Mondel objects is interpreted as a set of structured CPN tokens. A set of tokens is also a marking; i.e., a state. Tokens and markings are expressed in such a form that it is possible to obtain from a marking, by application of inference rules, executable transitions together with the corresponding successor markings. Therefore, from the initial object set, we can derive transition firing sequences or the whole reachability graph of the CPN associated with a Mondel specification.

5.4 Reflective Language Definition

In the formalism used to define the semantics of Mondel, the type definitions are static and used as templates for instance creation. Only the instances of a type are considered as objects. In order to support the construction of dynamically modifiable specifications, we need to have access to the type specifications during run time. A step in this direction is to consider types as objects and to enhance Mondel to provide means for altering type specifications. The principle of considering types as objects within a system, leads us to look at reflection and reflective architectures as a promising choice. The reflection principle has been studied for several object-oriented programming languages [Maes 87], [Ibra 88], [Coin 87], [Yone 89], [Male 90].

A language is called reflective if it uses the same internal structures to represent data and programs. In conventional systems, computation is performed only on data that represents entities of an application domain. In contrast, a reflective system must also contains some data which represent the structural and computational aspects of itself. We

designed RMondel, a reflective version of Mondel, by introducing a reflective semantics which extends the semantics described in Section 5.1.

Reflectivity in object-oriented languages is a natural consequence of the use of objects to represent the language constructs as well as the entities of an application domain. In the reflective version of Mondel, we adapt the reflection techniques [Maes 87] in a manner similar to ObjVLisp [Coin 87] and ObjVProlog [Male 90]. In contrast to the latter approaches, RMondel is strongly typed, supports object persistence, and considers not only types as objects, but also operations (methods), attributes and behaviors, as indicated by the instantiation and inheritance graphs of Figure 2. The instantiation graph represents the "instance of" relationship, and the inheritance graph represents the "subtype of" (inheritance) relationship. TYPE and OBJECT are the respective roots of these two directed graphs. The initial system state contains the kernel types: TYPE, OBJECT, AttributeDef, Operation and Behavior.

In order to keep a system in a consistent state, we defined [Erra 90] a set of constraints that must be satisfied by each type and its related types in the inheritance graph.These constraints define the consistency requirements of the type lattice which corresponds to the static semantics rules checked by the Mondel compiler. They are specified as invariants within the specification of the TYPE type.

The behavior part of a user object is defined by mean of objects which are specializations of the Statement type. Each action defined in a type is represented by an instance of one of the Statement type specializations which correspond to the different kinds of statements in the Mondel language [Erra 90]. The RMondel interpreter, specified in Mondel, is considered as an object which has a global view of the objects within the system. This interpreter object can select an object ready to perform a transistion, or a pair of objects involved in a rendez-vous, and then perform the corresponding transitions by changing the state of the selected object(s).

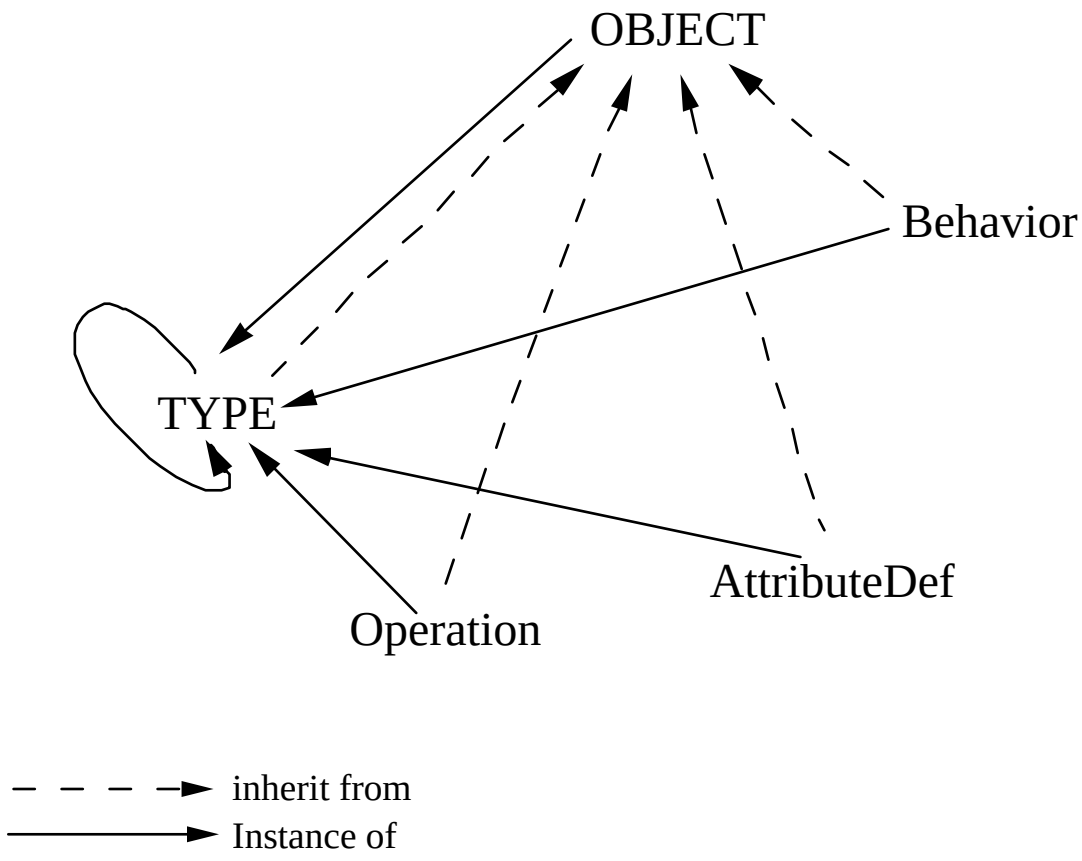

Figure 2. RMondel kernel types

6. Conclusions

The language Mondel presented in this paper belongs to the class of object-oriented languages, and shares many features with many other languages in this class (see for instance Table 1). These features were chosen in view of the intended application of Mondel as a specification language for describing real distributed systems and their control and management. The following features of Mondel are worth special mention, since they are not found in many other existing languages:

(1) Concurrent operations, not only among different objects, but also within a single object, which may be simultaneously involved in several actions.

(2) A Mondel specification has certain aspects of databases; in particular, persistent objects can be accessed through the equivalent of database queries. The concept of atomic transactions is also supported which makes it possible to provide distributed, fail-safe implementations of Mondel specifications by using standard fault recovery procedures developed for distributed databases.

(3) Communication among objects is synchronous. It is realized through the well-known concept of (remote) procedure call (RPC) with return parameters, which is used in most cases in a straightforward manner. However, when the alternatives in a CHOICE statement contain either calls of operations on other objects with output guards, or the acceptance of operations called by other objects with input guards, then the simple RPC concept allows the description of quite powerful synchronization requirements.

(4) Mondel has a formally defined semantics, which simplifies the construction of Mondel interpreters and compilers, and makes it possible to apply certain automated tools for the partial verification of Mondel specifications.

These features, together with multiple inheritance, strong typing and genericity, and an open-ended design to include assertional specifications of object properties, make Mondel a quite unique language.

We have used this language for several medium-size applications, including the description of fault management in a communication network [Leco 90], the OSI Reference Model [Mond 90b], and the OSI Directory service [Poir 90]. It is important to

note that the major difficulty in developing such formal descriptions is the structuring of the universe of discourse in the context where initially only vague and unstructured descriptions exist. We have therefore studied various design methodologies originally developed for database design or software engineering, and have adapted them into an object-oriented framework, as described in Section 2. The systematic application of such a methodology is helpful in organizing the system design, and ensuring its completeness.

We are presently working on formalizing the inheritance concept not only in respect to operation names and parameter type checking, but also in respect to the corresponding behavior definitions [Boch 89]. These issues have an impact on the maintenance and comparison of specifications, and must be considered in the context of environments for object-oriented system design and implementation. We are also working on a reflexive definition of Mondel, as described in Section 5.4, which makes it possible to describe an environment for the developement of Mondel specifications, together with the developed specifications, as a single system.

Acknowledgements

The authors thank A. Bean who contributed many interesting ideas to the design of the Mondel Language. They also thank A. Finkel and J. Lebensold for many fruitful discussions. The design methodology and the Mondel language described in this paper were developed within a research project jointly funded by Bell Northern Research and the Computer Research Institute of Montréal (CRIM). Financial support from the Natural Sciences and Engineering Research Council of Canada is also acknowledged.

Appendix: A Mondel Specification

```
unit DBaseExample use Predefined

invariant
        forall  p : Part  do
                assert  not  p.use(p);
        end;

type Part = Entity with
        name  :  String  key;
operation
        use   (p:Part)  :  Boolean   pure;
        cost            :  Integer   pure;
endtype Part

type CompositePart = Part with
        assembly_cost    :  Integer;
        mass_increment   :  Integer;
behavior exclusive
where

        procedure use (p:Part) : Boolean =
                ifexist u : UsedIn
                suchthat (u.composite = self)
                        and ( (u.component = p)  or  (u.component.use(p)) )
                then
                        return true;
                else
                        return false;
                end;
        endproc use
```

```
        procedure cost : Integer =
                define result = new Var[Integer] in
                        result := assembly_cost;
                        forall u : UsedIn
                        suchthat u.composite = self do
                                result := result^ + u.quantity * u.component.cost;
                        end;
                        return result^;
                end;
        endproc cost
endtype CompositePart

type BasePart = Part with
        purchase_cost   :  Integer;
        total_mass      :  Integer;
behavior exclusive
where

        procedure use (p:Part) : Boolean =
                return  false;
        endproc use

        procedure cost : Integer =
                return purchase_cost;
        endproc cost
endtype BasePart

type Supplier = Entity with
        name : String   key;
endtype Suppliers

type UsedIn = RelationShip with
        composite    :  CompositePart   key;
        component    :  Part            key;
        quantity     :  Integer;
endtype UsedIn
```

```
type Supplied = RelationShip with
        part  :  BasePart  key;
        by    :  Supplier  key;
endtype Supplied

type DBInterface = Entity with
operation
    select_base     (threshold  :  Integer)  :  Set[String]     pure atomic;
    compute_cost    (part_name :  String)   :  Integer          pure atomic;
behavior exclusive
where

        procedure select_base(threshold : Integer) : Set[String]  =
                define result = new Set[String] in
                        forall part : BasePart
                        suchthat part.cost >= threshold do
                                result!put(part.name);
                        end;
                        return result;
                end;
        endproc select_base

        procedure compute_cost(part_name : String) : Integer =
                ifexist  p:Part
                suchthat  p.name =  part_name then
                        return p.cost;
                else
                        -- error processing
                end;
        endproc compute_cost
endtype DBInterface

endunit DBaseExample
```

7. References

[Agha 87] G. Agha, C. Hewitt, "Concurrent Programming Using Actors" in: A. Yoneza, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, 1987.

[Amer 86] P. America, J. de Bakker, J. N. Kok, J. Rutten, "Operational Semantics of a Parallel Object-oriented Language", Proceedings of the 13th ACM Conference on Principles of Programming Languages, 1986.

[Amer 87] P. America, "POOL-T: A parallel Object-oriented Language" in: A. Yoneza, M. Tokoro (Eds.), Object-Oriented Concurrent Programming, MIT Press, 1987.

[Atki 87] M. P. Atkinson, O. P. Buneman, "Types and Persistence in Database Programming Languages", ACM Computing Surveys, Vol. 19, No. 2, June 1987.

[Bail 88] S. Bailin, "An Object-oriented Specification Method for ADA", ACM Proceedings of the Fifth Washington Ada Symposium, June 1988.

[Bail 89] S. Bailin, "An Object-oriented Requirements Specification Method", CACM, Vol. 32, No. 5, May 1989.

[Barb 90a] M. Barbeau, G. v. Bochmann, "Formal Semantics of Mondel", Progress Report Document No. 11 for CRIM/BNR Project, 1990.

[Barb 90b] M. Barbeau, G. v. Bochmann, "Formal Verification of Mondel Object-oriented Specifications Using a Coloured Petri Net Technique", in preparation.

[Barb 90c] M. Barbeau, G. v. Bochmann, "Verification of Lotos Specifications: A Petri Net Based Approach", Proc. of Canadian Conference on Electrical and Computer Engineering, Ottawa, 1990.

[Barb 90d] M. Barbeau, G. v. Bochmann, "Extension of the Karp and Miller Procedure to Lotos Specifications", Proc. of Computer-Aided Verification Workshop, New Brunswick NJ, 1990.

[Blac 87] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, "Distribution and Abstract Types in Emerald", IEEE Trans. on Soft Eng. Vol. 13, No.1, January 1987, pp. 65-76.

[Blah 88] M. R. Blaha, W. J. Premerlani, J. E. Rumbaugh, "Relational Database Design using an Object-Oriented Methodology", C. ACM, Vol. 31, No. 4, April 1988.

[Boch 89] G. v. Bochmann, "Inheritance for Objects with Concurrency", submitted for publication.

[Boch 90a] G. v. Bochmann, P. Mondain-Monval, L. Lecomte "Formal Description of Network Management Issues", Publication #734, Département d'IRO, Université de Montréal, 1990.

[Boch 90b] G. v. Bochmann, P. Mondain-Monval, S. Poirier, L. Lecomte, M. Barbeau, N. Williams, "System Specification with Mondel and Relation with Other Formalisms", CRIM/BNR Project Progress Report No. 13, 1990.

[Boch 90c] G. v. Bochmann, "Protocol Specifications for OSI", Computer Network and ISDN Systems, April 1990.

[Boch 90d] G. v. Bochmann, A. Finkel, "Impact of Queued Interaction on Protocol Specification and Verification", Proc. Intern. Symp. Interoperable Inf. Systems (ISIIS), Nov. 1988, Tokyo, pp. 371-382.

[Bois 89] H. Bois, "Une méthode de développement de logiciels fondée sur le concept d'objet et exploitant le langage ADA", Thèse de Doctorat, Université Paul Sabatier, Toulouse, France, October 1989.

[Booc 86] G. Booch, "Object-Oriented Development", IEEE TSE, February 1986.

[Bray 85] G. Bray, D. Pokrass, "Understanding ADA", John Willey and Son, 1985.

[Buck 83] G. N. Buckley, A. Silberschatz, "An Efficient Implementation for the Generalized Input-output Construct of CSP", ACM Tr. on Progr. Lang. and Systems, Vol. 5, No. 2, April 1983, pp. 223-235.

[Card 88a] L. Cardelli, "A Semantics of Multiple Inheritance", Information and Computation 76, 1988, pp. 138-164.

[Card 88b] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson, "Modula 3 Report", D. E. C., 1988.

[Chen 76] P. P. Chen, "The Entity-Relationship Model - Toward a Unified View of Data", ACM Trans. on Database Systems, Vol. 1, No. 1, March 1976, pp. 9-36.

[CCITT 87] CCITT, "Specifiction and Description Language SDL", Recommendation Z.100, 1987.

[Coin 87] P. Cointe, "Metaclasses are First Class: The ObjVLisp Model", OOPSLA'87, ACM Sigplan Notices, Vol. 22, No. 12, pp. 156-167.

[Erra 90] M. Erradi, G. v. Bochmann, "Definition de RMondel", in preparation.

[Ehri 85] H. Ehrig, B. Mahr, "Fundamentals of Algebraic Specification 1", Springer-Verlag, Berlin, 1985.

[Gao 89] Q. Gao, G. v. Bochmann, "Distributed Implementation of Lotos Multi-rendezvous", Proceedings of PSTV IX, Enschede, 1989.

[Gard 89] G. Gardarin, P. Valdurief, "Relational Databases and Knowledge Bases", Addison-Wesley, 1989.

[Gara 89] H. Garavel, E. Najm, "Tilt: From Lotos to Labelled Transition Systems", in: P. H. J. van Eijk, C. A. Vissers and M. Diaz (Eds.), The Formal Description Technique Lotos, North-Holland, 1989.

[Gold 83] A. Goldberg, D. Robson, "Smaltalk-80 - The Language and its Implementation", Addison Wesley, 1983.

[Gotz 86] R. Gotzhein, "Specifying Abstract Data Types with Lotos", Proc. IFIP Workshop on Protocol Specification, Testing and Verification, VI, North Holland Publ., 1986, pp. 15-26.

[Gray 81] J. Gray, "The Transaction Concept: Virtues and Limitations", Proc. IEEE Conf. on VLDB, Cannes, Sept. 1981, pp. 144-154.

[Hoff 88] D. Hoffman, R. Snodgrass, "Trace Specifications: Methodology and Models", IEEE Trans. on SE, Vol. 14, No. 9, Sept. 1988, pp. 1243-1252.

[Ibra 88] M. H. Ibrahim and F. A. Cummins, "KSL: A Reflective Object-Oriented Programming Language", IEEE, Proceedings of the Int. Conf on Computer Languages, 1988, pp. 186-193.

[ISO 87a] ISO/TR 9007, "Concepts and Terminology for the Conceptual Schema and the Information Base", 1987.

[ISO 87b] ISOTC 97/SC 6, "Specification of Abstract Syntax Notation One", IS 8824, 1987.

[ISO 88a] ISO, "Information Processing Systems - Open Systems Interconnection - ESTELLE - A Formal Description Technique Based on an Extended State Transition Model", IS 9074, 1988.

[ISO 88b] ISO, "Information Processing Systems - Open Systems Interconnection - LOTOS - A formal Description Technique Based on the Temporal Ordering of Observational Behavior", IS 8807, 1988.

[ISO 89a] DP 10165-2, "Systems Management - Object Management Function".

[ISO 89b] DP 10165-2, "Structure of Management Information - Definition of SupportObjects".

[ISO 89c] DP 10165-3, "Structure of Management Information - Definition of Management Attributes".

[ISO 89d] DP 10165-4, "Structure of Management Information - Guidelines for Managed Object Definition".

[Jalo 89] P. Jalotte, "Functional Refinement and Nested Objects for Object-orientedDesign", IEEE TSE, Vol. 15, No. 3, March 1989.

[Jens 87] K. Jensen, "Coloured Petri Nets", in W. Brauer et al. (Eds.): Petri Nets - Central Models and their Properties, LNCS 254, Springer, 1987.

[Kowa 79] R. Kowalski, "Logic for Problem Solving", The Computer Science Library, 1979.

[Leco 89] L. Lecomte, "Sur la compilation d'un langage orienté-objet", M.Sc. Thesis, Université de Montréal, 1989.

[Leco 90] L. Lecomte, G. v. Bochmann, P. Mondain-Monval "Un modèle orienté objet pour le système de transmission FD-565", CRIM/BNR Progress Report Document No. 8, 1990.

[Lipp 89] S. B. Lippman, "C++ Primer", Addison-Wesley, 1989.

[Lisk 87] B. Liskov, D. Curtis, P. Johnson, R. Scheifler, "Implementation of Argus", Proceedings of the 11th Symposium on Operating Systems Principles, 1987.

[Lisk 88] B. Liskov, "Distributed Programming in ARGUS", C. ACM, Vol. 31, No. 3, March 1988.

[Maes 87] P. Maes, "Concepts and Experiments in Computational Reflection", OOPSLA'87, ACM Sigplan Notices, Vol. 22, No. 12, pp.147-155.

[Male 90] J. Malenfant, "Conception et implantation d'un langage de programmation logique, par objets et repartie", Ph.D. Thesis, Département d'IRO, Université de Montréal, January 1990.

[Marc 89] S. Marchena, G. Leon, "Transformation from Lotos Specs to Galileo Nets", in: K. J. Turner (Ed.), Formal Description Techniques, North-Holland, 1989.

[Meye 86] B. Meyer, "Genericity Versus Inheritance", OOPSLA'86 Proceedings, 1986.

[Meye 87] B. Meyer, "Reusability: the Case for Object-oriented Design", IEEE Software, March 1987.

[Meye 88] B. Meyer, "Object-oriented Software Construction", C. A. R. Hoare Series Editor, Prentice Hall, 1988.

[Miln 80] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in CS, No. 92, Springer Verlag, 1980.

[Mond 90a] P. Mondain-Monval, G. v. Bochmann, "An Object-oriented Software Design Methodology", Progress Report Document No. 7 for CRIM/BNR project, June 1990.

[Mond 90b] P. Mondain-Monval, G. v. Bochmann, "An Object-oriented Software Architecture for the OSI Basic Reference Model", Publication #736, Département d'IRO, Université de Montréal, 1990.

[Muel 83] E. T. Mueller, J. D. Moore, G. J. Popek, "A Nested Transaction Mecanism for LOCUS", C. ACM, 1983.

[Mura 89] T. Murata, B. Shenker, S. M. Shatz, "Detection of Ada Static Deadlocks Using Petri Net Invariants", IEEE TSE, Vol. 15, No. 3, 1989, pp. 314-326.

[ODP 88] Working Document on Topic 6.1 - "Modeling Techniques and their Use in ODP", ISO/IEC JTC1/SC21 N 3196, December 1988.

[Parn 72] D. Parnas, "On The Criteria to be Used in Decomposing Systems Into Modules", C. ACM, Vol. 15, No. 2, 1972, pp. 1053-1058.

[Poir 90] S. Poirier, "Evaluation du langage de spécification Mondel à la description de protocoles de communication", M.Sc. Thesis, Université de Montréal, 1990.

[Plot 81] G. D. Plotkin, "A Structural Approach to Operational Semantics", Aarhus University, Report DAIMI FN-19, 1981.

[Shat 88] S. M. Shatz, W. K. Cheng, "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior", The Journal of Systems and Software, Vol. 8, No. 5, 1988, pp. 343-359.

[Suzu 90] S. Toshinori, S. M. Shatz, T. Murata, "A Protocol Modeling and Verification Approach Based on a Specification Language and Petri Nets", IEEE TSE, Vol. 16, No. 5, May 1990, pp. 523-536.

[T1M1 89] Committee T1-Telecommunications Standards Contribution - "Modelling guidelines", February 1989

[Trip 89] A. Tripathi, E. Berge, "An Implementation of the Object-oriented Concurrent Programming Language SINA", Software-Practice and Experience, Vol. 19, No. 3, March 1989, pp. 235-236.

[Viss 88] C. Vissers, G. Scollo, M. v. Sinderen, "Architecture and Specification Style in Formal Descriptions of Distributed Systems", Proc. IFIP Symposium on Prot. Spec., Verif. and Testing, Atlantic City, 1988.

[Ward 89] P. T. Ward, "How to Integrate Object orientation with Structured Analysis and Design", IEEE Software, March 1989.

[Will 90] N. Williams, G. v. Bochmann, "Description technique d'un simulateur pour le langage Mondel", Progress Report Document No. 10 for CRIM/BNR Project, 1990.

[Yone 89] A. Yonezawa, T. Watanabe, "An Introduction to Object-based Reflective Concurrent Computation", ACM Sigplan Notices, Vol. 24, No. 4, 1989, pp. 50-54.