Steps Towards a Method for the Formal Modeling of Dynamic Objects

Roel Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081A
1081 HV Amsterdam

uucp: roelw@cs.vu.nl

ABSTRACT

Fragments of a method to formally specify object-oriented models of a universe of discourse are presented. The task of finding such models is divided into three subtasks, *object classification*, *event specification*, and the *specification of the life cycle* of an object. Each of these subtasks is further subdivided, and for each of the subtasks heuristics are given that can aid the analyst in deciding how to represent a particular aspect of the real world. The main sources of inspiration are Jackson System Development, algebraic specification of data- and object types, and algebraic specification of processes.

Keywords: Formal specification, Object-oriented modeling methods.

1. Introduction

1.1. The problem

Conceptual modeling is the process of finding an explicit model of a universe of discourse (UoD). Other terms used for ‘‘conceptual modeling’’ are ‘‘information analysis’’ and ‘‘business area analysis’’. The result of this is a conceptual model (CM) of the current or of desired versions of the UoD. If the desired versions contain automated systems, the CM of that version of the UoD serves as an input to the DB *design* process, in which, among others, a DB schema is defined.

In this paper, we discuss *object-oriented* conceptual modeling. The defining characteristic of this kind of modeling is that it produces object-oriented CM’s of a UoD, just as entity-relationship modeling [17] produces ER models of a UoD and NIAM [48] produces NIAM models of a UoD. In particular, we discuss the following questions.

1. What is the common core of object-oriented CM’s?
2. How should object-oriented CM’s be specified formally?
3. Which development method is best suited to object-oriented CM’s?

We briefly discuss each of these questions in turn.

First, object-oriented DB research has started as a set of techniques to combine the advantages of object-oriented languages with database techniques [70]. This has resulted in a number of experimental object-oriented DB systems [6, 21, 29, 40]. Only recently, there are attempts to define a common core to object-oriented DB’s [2, 70], which converge on concepts like object identifier and the encapsulation of a local state in an object. These characterizations are not yet formal and many of

their features are not yet generally accepted. Also, they are about object-oriented *databases* rather than object-oriented *conceptual models.* I will give my view on what the common core of object-oriented CM's is in section 2 below, and contrast it briefly with object-orientation in the areas of DB's.

With regard to the second question, how object-oriented CM's should be specified formally, the matter is even more open. The current approaches to formalizing object-oriented models can be classified as logic-based [7, 36, 69] and algebra-based [20, 23, 24, 52, 53]. In this paper, I follow the algebraic route, based on work done in the past few years [59]. This combines process algebra [8, 9, 10] with the algebraic specification of abstract data types in order to specify dynamic abstract objects. The CM specification language resulting from this combination is called CMSL. The first version of CMSL is described in [60]. The second version is described in [64] and will be used for illustrative purposes in this paper.

The third question concerns the development method best suited for object-oriented CM's. Research into this question still has to get off the ground, and it is the main aim of this paper to develop elements of an object-oriented conceptual modeling method. We use the modeling stage of Jackson System Development (JSD) to give us clues on tasks and heuristics to be executed during object-oriented conceptual modeling. However, the representation techniques of JSD are found to be inadequate, so we use our own techniques, introduced in [59]. In addition, JSD is primarily process-oriented and has no means to model static structures like generalization and aggregation, and we add these as elements to the modeling method as well. Integration of these structures in a single CM is formally defined in [59]. In this way, we hope to make a significant step in the direction of an object-oriented conceptual modeling method, without claiming to present a final product.

In the next subsection, we give a very brief introduction to JSD, to make plausible that is is a reasonable starting point for an object-oriented method. It should also be clear that it is a starting point, not the final answer. Next, section 2 defines our view of object-orientation in the field of conceptual modeling. Sections 4, 5 and 6 then each discuss one aspect of object-oriented conceptual modeling. In section 7 we make a short remark on the way the different tasks in a modeling method can be ordered, and section 8 summarizes the paper and discusses topics of current research.

Throughout the paper, we highlight important principles by placing them in boxes. These are either definitions of concepts, or axioms that limit the kinds of CM we consider, or consequences of the definitions or axioms previously stated as principles. The definitions and axioms are motivated by reference to existing CM structures or by logical analysis of the kinds of UoD we want to consider. The reason for calling them principles rather than axioms, definitions or consequences is, first, that the treatment is not so rigorous that they deserve these names, and second, that they can all be used to help resolve decisions about what to include in a particular CM and what to exclude from it.

1.2. Jackson system development

JSD divides system development into three major tasks, called the modeling, network and implementation stage, each of which is divided into a number of steps [15, 32, 54].

1. The goal of the *modeling stage* is to build a model of the UoD. Steps to achieve this goal include finding the entities to be modeled, the actions suffered or performed by these entities, their possible communications, and their possible life cycles.
2. The goal of the *network stage* is to define the functions of the IS to be implemented. Where the modeling stage describes the current UoD, the network stage concentrates on the desired functions of an IS. Steps towards reaching this goal include connecting the model to the UoD and specifying the reports that the IS is to make to the user, the input the IS requires from the UoD, and feedback loops from the IS to itself that will automate some of the tasks currently done in the UoD.
3. The goal of the *implementation stage* is to produce an implemented system. Steps towards this goal include mapping of the potentially infinite number of process on a finite number of processors (usually one), specifying a scheduler to monitor these processes, defining files that will contain the state vectors of parallel processes, and specifying detailed operations.

This method is object-oriented in a weak sense because it specifies a UoD-oriented model in the modeling stage. In a stronger sense, it is object-oriented because it builds a model around the concept of an object (‘‘entity’’ in JSD), which encapsulates behavior. However, the static entity structures in JSD are rather weak, and as said before, the representation techniques have certain inadequacies. In what follows we will therefore take JSD as our starting point, but will pick and choose from it what we need and combine it with whatever other material that seems suitable.

2. Object-oriented conceptual models

2.1. UoD-orientation

We define the UoD as the set of all actual and possible entities of interest, and a CM of the UoD as an abstract representation of the UoD. The thing to be noted about this is that

CM’s are UoD-oriented.

Principle 1.

This distinguishes them from data models, which are computer-oriented (albeit at an implementation-independent level), because they are about data in a computer.

We distinguish two kinds of entities, objects and masses. An *object* is anything that can be put in a set. Because we can talk meaningfully about a set of (actual or possible) people, projects, companies, and trees, these entities are objects. Another way of putting this is that *objects can be counted.* This does not mean that a set into which an object is put is countable, but simply that it has a cardinality. This property of objects is important, for counting is an important DB application.

By contrast, an entity *e* is a *mass* iff it cannot be counted, i.e. iff it cannot be put into set. Water, profit, money deposits, weight, gold, and air are all examples of masses. Of masses we can ask how *much* of it there is, whereas by contrast, of objects we can ask how *many* of them there are.

Masses are never represented in DB’s or CM’s. Instead, for every kind of mass about which we want to represent information, we define a *measure,* which is a function from masses to discrete objects [56]. For example, profit is measured in terms of a money unit, weight in a weight unit, etc. We then represent how many units go into a certain mass, i.e. we represent an abstract object which represents how much of the mass there is.

2.2. Object-orientation

Because we only represent objects in a CM, we conclude that

all CM’s are object-oriented in the general sense.

Principle 2.

By this we mean that all CM’s represent possible objects and not possible masses. This holds for hierarchic, network, relational, and semantic models, so the question arises what is distinctive about the type of model currently called object-oriented. We explain this by using an old philosophical controversy, concerning the relation between an object and its properties. The common starting point of this controversy is the fact that objects have properties. The difference of opinion is about whether there is something, often called a *bare particular,* which underlies the properties of an object, or whether an object is just a bundle of properties [38]. A bare particular is not a property, but that to which a property is attached. Bare particulars can explain that objects may have identical properties but yet be "numerically different", i.e. be counted as different. This is important, because, as we saw, objects are characterized by the fact that they can be counted. The philosopher Loux [38] argues that the view of objects as just bundles of properties cannot account for the fact that objects may be indistinguishable (have the same properties) but yet be numerically different. The only way the bundle theorist could account for this is by assuming that the two object differ in at least one property. But that would contradict the assumption that the objects are indistinguishable, for indistinguishability implies identity of properties.

We are not interested in the metaphysical truth of either position, but in the usefulness of each position for conceptual modeling. Relational models represent objects as bundles of properties, and so cannot represent the difference of objects which have *all* their properties identical. Object-oriented models, in my view, are characterized by the fact that they use globally unique *object identifiers* to distinguish numerically different objects. Identifiers are globally unique, because they must account for numerical difference of any pair of different objects. They thus function as bare particulars. This is useful for conceptual modeling, because a CM usually abstracts many details of the UoD. In this abstraction, the difference between the properties of objects may be lost, so that we will need object identifiers to keep indistinguishable objects apart.

> A CM is *object-oriented in the strict sense* [1] iff every object has a globally unique object identifier.

Principle 3.

Object identifiers are not only an old idea from philosophical logic, they are also an old idea in data modeling. Hall et al. [25] introduced the concept of a surrogate in 1976 and Codd [18] used in is the system RM/T. Khoshafian and Copeland [35] coined the term ''object identity'' and summarized the arguments for it. Since then, the concept is listed as one of the essential elements of object-orientation [2], but is it sometimes also called ''object identifier.'' We follow this latter practice, because object identity is a relation between objects and is therefore a less suitable concept.

2.3. Object identifiers

Suppose an object has a *proper name* n, and that it has the same proper name in all possible states of the UoD. In technical terms, n is a *rigid designator* for the object it names [37]. Because the object has by definition the same identifier in all possible states of the UoD, we can also view n as the name of the object identifier. However, there is a difference between the two, because the identifier is defined to be an ingredient of the object, and n is a part of speech that refers to the identifier.

Now additionally assume that all different possible objects have different proper names. Then there is a 1-1 correspondence between proper names and object identifiers. This creates an even closer connection between object identifiers and proper names.

Finally, we abstract from the physical nature of the UoD and go to the abstract CM. At that level of abstraction, we may just as well identify object identifiers and their proper names. So in the CM, the proper name of an object *is* an ingredient of the abstract objects living in the CM. This brings us to Herbrand models and quotient-term algebras as natural candidates for formalization of the CM. In both kinds of formal models, we use constants (i.e. proper names) as the material out of which the model is built. Because object identifiers must account for identity and difference of objects, the concept of *equality* is all-important in object-oriented CM's, and we use the algebraic quotient-term construction in [59].

Object identifiers not only account for the numerical difference of objects, they also account for their *persistence* through change of properties. It makes sense, in an object-oriented CM, to say that I am the same person as I was 30 years ago, because I have the same identifier as 30 years ago, though I may have none of the properties I had back then.

A third property of identifiers is that they have a *type.* If I want to count objects, I count identifiers, but this presupposes that they have a type. For example, a set of three passengers of a bus may be a set of 1 person, if that person entered the bus three times. With each type come criteria of identity, which determine when instances of the type are the same or different. Put differently, a query how many passengers have been on the bus should not answer ''50'', but ''50 persons'', or ''50 passengers'', or in general ''50 τ'' for some type τ. So every identifier is an *instance* of at least one type.

---

1. We omit this qualification from now on. So ''object-oriented'' without further qualification means ''object-oriented in the strict sense''.

Finally, identifiers may be *structured.* For example, a set of three person identifiers is itself the identifier of a higher-order object with properties like average age, maximum age, etc. When a fourth person identifier is added to this set, a different higher-order object ensues, identified by a set of 4 identifiers which differs (by set-equality) from the previous one. We make no decision at this point on the kinds of composition allowed, but we will at least allow that object identifiers can be composed of other object identifiers.

To sum up, object identifiers have the following tasks.

1. Objects are numerically different iff they have different identifiers.
2. Identifiers persist through change of properties.
3. An identifier is an instance of at least one type.
4. Identifiers may be composed of other identifiers.

Principle 4.

Note that at this level of abstraction, object identifiers are not an implementation concept like surrogates or Codasyl global database keys. Availability of object identifiers in the specification language is therefore not an availability of an implementation element in the specification. We will make object identifiers available in the specification language only subject to the logic of identifiers, summarized in principle 4 below and formalized in the specification language we use for illustration, CMSL.

It is interesting to compare this with the use of identifiers in programming languages. In programming languages, identifiers are names of variables, constants, procedure and function names, etc. Looking at the use of identifiers as names of variables, we can treat the *value* of a variable as the *property* of an identifier. Following the list of characteristics of identifiers in principle 4, identifiers in programming languages serve 1. to keep indistinguishable values numerically apart, 2. to preserve the identity of variables through change of value, 3. are always declared as having a type, and 4. can be constructed from more primitive identifiers. These are thus fundamental properties of all identifiers, in programming languages as well as in CM's. The important difference between the use of identifiers in programming languages and in CM's is that they are usually part of the specification in the first, but part of the model in the second. By making identifiers semantic, it is possible to generate infinitely many possible identifiers in the CM, as opposed to a programmer declaring finitely many identifiers in a program. In this respect, there is a close relationship between pointers in programming languages and identifiers in object-oriented CM's. Pointers generated by a program function much like identifiers generated in an object-oriented model.

So far, an object has been analyzed to consist of an identifier and a set of properties. We will not define what properties are in this paper. They include at least attributes like age and name, but will also include events like move and increase of age and roles like employee and student. In general, objects found in the real world are dynamic, and so our CM should be able to represent object dynamics. That the real world changes is so obvious that it should hardly need to be mentioned. Because a CM is UoD-oriented, it should also be obvious that objects in a CM should be dynamic.

Many abstract objects, like numbers, strings and Booleans, are not dynamic but live eternally and unchanging in an abstract world. This, too, should be represented by the CM.

An important principle implicit in the above account of object-orientation is that objects *encapsulate* a state and behavior. This is so to the extent that real-world objects have a state, execute events, and go through a life-cycle of events independently of other objects. To the extent that there is a connection between the state and behavior of different objects, encapsulation is transgressed. The aim of object-oriented modeling is to define precise and limited interfaces of objects through which they can communicate with other objects.

3. Object-oriented conceptual modeling

After discussing the nature of object-oriented conceptual models, we turn to the process of object-oriented conceptual modeling. This is an exercise in methodology, construed as the study of methods. Before we dive deeper into this, we make a few remarks on how to describe methods.

Any CM is an abstraction of a UoD, and we consider only CM's that are specified linguistically by a *CM schema* (CMS, see figure 1).

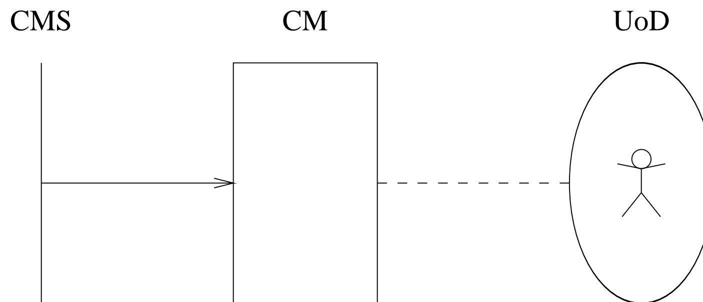

Figure 1.

We require the CMS to have a formally defined relation to the CM. For example, the CMS may be a theory in a formal language with the CM as a model, or it may be a formula in a language denoting a particular element of a model. This is indicated by the black arrow. The dashed line indicates that the CM is an abstraction of the UoD.

Any development method offers some specification language for the CMS, some set of structures which can be represented by the CM, and some practical advice about how to proceed from the initial knowledge of a UoD to a specification of a CM. So we have

(1) method = specification language + model structures + practical advice.

By ''practical advice`` we mean a set of tasks to be executed, a preferred order of executing these tasks, and a set of heuristics which give guidelines on how to make decisions when executing these tasks. Thus,

(2) practical advice = tasks + ordering of tasks + heuristics.

The task ordering is often presented as a linear, total ordering. A task is usually presented as a series of steps or subtasks, aimed at achieving a certain goal:

(3) task = goal + subtasks.

The breakdown of tasks can be continued down to the level of detail one wants. An important part of the goal of a task is whether it concerns the current state of the UoD or some possible desired state.

This way of describing methods contains already one important characteristic of object-orientation, UoD-orientation. Part of what object-oriented methods should contain in addition to this has already been stated: the CM produced by such a method should contain discrete objects with globally unique object identifiers. All other properties of objects (attributes, events, constraints, life cycles) should be specified as properties attached to an object identifier. Only where the UoD itself has no modular structure, should a CM of the UoD lack a modular structure.

Apart from UoD-orientation and a modular structure organized around object identifiers, there is a third characteristic of object-oriented development. This is that our specification of objects precedes the specification of the functions of the information system that will be implemented. As Booch [12] rightly remarks, functional decomposition of a problem emphasizes algorithmic abstractions and is silent about the agents that suffer or perform the actions specified in the algorithms. Moreover, I would add that functional decomposition is oriented towards implementation, which is where the algorithms will be made to live, rather than towards the UoD, which is where the objects live. Even where there is an overlap between the UoD and the implementation, such as in the implementation of a clock in a UoD, the difference between implementation-orientation and UoD-orientation remains, and remains important.

If these are the characteristics of object-oriented modeling, what are the reasons why one should follow such a method (assuming such a method has been specified)? Each of the three characteristics contains reasons for choosing an object-oriented method. *First,* UoD-orientation allows the analyst to get sufficient knowledge of the UoD to avoid serious errors in a later stage of development. This is received wisdom from structured analysis methods [19]. *Second,* object-orientation in the strict sense allows one to represent indistinguishable objects, object dynamics, object classification, and complex and higher-order objects in a natural way. (See principle 4 and [59] for an elaboration of these points.) Moreover, as Booch remarks, concentrating on the objects underlying the processes in the UoD provides one with a natural principle of modularization of the software to be implemented. *Third,* putting model specification before function specification results in a more stable IS. This is one of the main motivations behind Jackson System Development (JSD) [14, 32]. The functions of an IS change as fast as the requirements or even the desires of its users, and are therefore not a solid base on which to build an IS. A model of the UoD, on the other hand, is as stable as it should be and, if it represents all the relevant facts about the UoD, can serve as a basis upon which to implement a variety of functions not thought of during modeling. This does not mean that the UoD never changes, but it does mean that we assume that the UoD changes less frequently than the functions of the information system.

In the following three sections, we use the modeling stage of JSD to define fragments of an object-oriented modeling method. The JSD modeling stage is subdivided into a number of steps, in which the relevant actions in the UoD are specified, the entities suffering or performing these actions are specified, and the actions of an entity are composed into the life cycle of the entity. We call these steps *object classification, event specification,* and *process specification.* Each of these is discussed in a section. At the end of each section, we summarize the semantic structures specified in the step, give an example of a specification of these structures in CMSL, and summarize the tasks and heuristics required to produce this specification.

4. Object classification

4.1. Objects

We already encountered two principles that can guide one in deciding which objects there are in the UoD. The first principle is that the world is partitioned into two kinds of entities, masses and objects. These can be distinguished in two ways, viz.

1. objects are discrete, and
2. splitting or merging a mass yields a mass of the same type.

Principle 5.

We simply assume here that these two criteria are equivalent. ''Splitability'' implies non-discreteness, and non-discreteness implies ''splitability.'' This is a simplifying assumption, and in the future we plan to look closer into this matter.

The second principle we already encountered is that

all objects have a globally unique identifier.

Principle 6.

These principles are used in JSD to decide what are the JSD entities (i.e. objects) in the UoD. Jackson [32, p. 66] says that a JSD entity must be capable of being regarded as an individual, and, if there is more than one entity of a type, of being uniquely named.

4.2. Types

To distinguish objects in the UoD is to distinguish *classes* of objects. To point at an object (which is what an object identifier does) is to point at an instance of one or more *classes* of objects, e.g. an employee, car, chair, bank account, etc. We start by fixing some terminology concerning classes, and we do this by borrowing some distinctions made in philosophy. The distinction between classes and

instances has kept philosophers busy since Aristotle [16, 39, 38, 68]. This has led to a different terminology than we are used in computer science, but the distinctions can be immediately applied to conceptual modeling. What is called a *type* in computer science is called a *universal* in philosophy, and what is called an *object* in this paper, comes close to what is called a *particular* in philosophy. (Object *identifiers* correspond to *bare* particulars.) Universals or types have the following two characteristics [68, p. 65].

1. First, a type is distinguished from a particular by the fact that it can be *predicated* of things. For example, `red` is a type because we can say (truly or falsely) of something that it is red. Anything of which a type τ can be predicated is called an *instance* or *exemplar* or *example* or *case* of τ. A type may be an instance of another type. For example, `red` is an instance of `color`. The characteristic of types is that they have instances, i.e. can be predicated of things. The characteristic of particulars is that they have no instances, i.e. cannot be predicated of anything. For example, `Amsterdam` cannot be predicated of anything; it is just itself. Apparently, instantiation is a hierarchy, in which at each level, a type can be instantiated as lower-level types, and in which the bottom level consists of particulars.
2. Second, a type can be predicated of *many* things. So `being Amsterdam` is not a type, because it can be predicated of only one thing, Amsterdam. A type should represent what is common to many cases, not what is peculiar to only one particular.

We transfer this to the context of conceptual modeling by reading ''object'' for ''particular''. So objects are the bottom-level of the instantiation hierarchy. We furthermore assume that each object is an instance of at least one type. Because there is a 1-1 correspondence between objects and object identifiers, this means that each object identifier is an instance of at least one type, as was already stated in principle 4. Assuming the instantiation relation as an unexplained primitive, we have that

object identifiers are distinguished from types in that they have no instances.

Principle 7.

On the other hand, types themselves may be instances of higher-order types. A type which is an instance of a higher-order type is called a *generic entity* in JSD [32, p. 72]. For example, in an inventory database we may represent the number of widgets in store as a generic entity `WIDGET` with an attribute `number`. We may run into problems if we want to integrate this database with a quality control application, which represents particular widgets with attribute `serial-number` (cf. [34, p. 2-3]). The problem is that `WIDGET` is really a type in one database and an instance in the other, and that no means has been provided to treat a type as an instance of a higher-order type.

The concept of a type is *intensional,* because it concerns the common features of a large set of cases. We define a *class* to be the *extension* of a type, which is the set of all possible instances of a type, whether or not they exist. The set of instances existing in a particular state of the UoD is called the *existence set* of a type in that UoD state.

4.3. Structured identifiers

Principle 4 mentions one more task of identifiers, that of representing structural relations between objects. As a matter of fact, `WIDGET` is an example of an identifier composed of other identifiers through the *instance-of* relation. Another example is the set of employees in a department. Let $p_1, \ldots, p_n$ be the object identifiers of employees in a department, then the set $\{p_1, \ldots, p_n\}$ identifies, is the identifier of, the object consisting of these employees. This higher-order object may have attributes `average-age`, `max-salary`, and `dept`. (Since all elements of the set by definition have the same department, this is a collective attribute.) In JSD, such an entity is called a *collective entity* [32, p. 72]. In CMSL, this is generalized to the concept of *structured identifier,* which is an identifier related to component identifiers of another type through any valid data type constructor (e.g. a set constructor, or a tuple or string constructor, etc.). An object with a structured identifier is also called a *higher-order object.*

Usually, one would want to require that the components of the identifier of an existing higher-order object themselves exist. We will not go into the topic of existence constraints here, because

that would require a paper in itself.

Structured and unstructured identifiers can both be used as attribute values. An `employees` attribute of departments, whose value holds the identifiers of the employees currently in the department, has a structured identifier as value. An object with a structured identifier as possible attribute values is called a *complex object* in CMSL.

4.4. Natural kinds and roles

All instances of a type share a common structure. This common structure is laid down, for example, in the definition of the type, as the set of applicable attributes or events or as the set of constraints applicable to these attributes or events. In this way, we deal with the infinite by finite means. The set of possible objects is usually infinite, and the set of all possible properties of each object is usually infinite as well. The only way the human mind can deal with this is to state finite generalizations about these infinite sets.[2]

There are three important properties of classifications. First, each class should be *natural.* This is often expressed by saying that a (natural) class should "divide at the joints", or that the chance of finding generalizations that are true of all and only the instances of a natural class is large [45, pp. 465 ff.]. We express this by saying that

> a natural class is the largest set of instances about which a set of empirical generalizations are true.

Principle 8.

For example the set of persons, in a particular CM of a UoD, may be the largest set of instances that have a name and have an age under 120. That this is a natural class appears from the fact that a lot of other generalizations are also true of this class. For example, they are also the largest set of instances with an address, a birthplace, a country of origin and a nationality, and there is a systematic relation between the birthplace and the country of origin.

The second important property of natural classes is that they exist in ''nature'' as the information analyst finds it, i.e. in the UoD. They are not invented by the information analyst, and he or she can be wrong about the definition of a natural class. This is so even if the classes are artificial, like the class of vehicles, cars, buildings or companies. As far as the information analyst is concerned, it is a fact of nature that these classes exist in the UoD.[3]

The third and final property of natural classes is that they are either essential or contingent. By saying that a natural class is *essential* we mean that an object that is an instance of the class, is an instance of the class in all possible states of the UoD. This means that the common structure associated with the class is an essential feature of instances of the class. An object that is an instance of the class cannot exist without displaying those features. For example, a person cannot exist without being a person, so the class of persons is essential. Other examples of essential classes are the class of houses, cars, vehicles, etc.

By saying that a natural class is *contingent* we mean that an object that is an instance of the class, may, in other states of the UoD, fail to be an instance of that class. So an object that is an instance of such a class can exist without being an instance of that class. For example, a student may exist without being a student, so the class of students is contingent. Other examples are employees, owners, spouses, etc.

Essential natural classes are called *natural kinds* in the philosophical literature [49, 51], and we

2. This is in general the reason for classifying the objects in a UoD. For example, of the six different purposes of biological classification listed by Warburton [55], the specification of classes about which we can make empirical generalizations is the primary one. In biology, the situation is compounded by the fact that classes are also used as units of evolution, and that as such their elements need not share a common structure. Instead, they behave as a unit reproductively. See for example [22, 27, 30, 31].

3. This contrasts with Schwartz [50], who calls artificial classes nominal kinds. He takes the standpoint of a subject living in the UoD, not of a supposedly objective analyst outside the UoD.

will use that term here as well. So

> an instance of a natural kind cannot exist without being an instance of that kind.

Principle 9.

Because any object has at least one type, we conclude that any object has at least one natural kind which it is an instance of.

We will call contingent classes *roles,* and an instance of a role is said to *play* the role. So we have that

> a role instance can exist without playing that role.

Principle 10.

For each role there should be events which cause an object to start playing the role and/or to stop playing it. This shows also that each object playing a role should be of a natural kind, for otherwise it could not exist without playing that role. Remember that any object that possibly exists is an instance of at least one natural kind. This means that each role must be a subclass of at least one natural kind, which is the kind of objects able to play that role. For example, the student role is a subclass of the natural kind of persons. Furthermore, a natural kind cannot be a subclass of a role. Otherwise, an instance of the role could not exist without being an instance of the role, contradicting principle 10. For example, it would be contradictory for the natural kind `PERSON` to be subclass of the class `SPOUSE`, for then all persons could not exist without being spouses.

> 1. A role must be a subclass of at least one natural kind.
> 2. No natural kind is a subclass of a role.

Principle 11.

A detailed discussion of the constraints on the relations between natural kinds and roles is given in [59]. The formalization of natural kinds and roles given there is briefly summarized in section 5.5. below.

The concept of a role has surfaced several times in data modeling research, but often with meanings unrelated to the one in which we use it here. The closest comes the role concept of Bachman & Daya [3], who define a role as a behavior pattern that may be assumed by entities of different kinds. They define *essential roles* as we defined natural kinds, i.e. an entity of such a type cannot exist without being of that type [3, p. 466]. They also allow roles that have more than one natural kind as supertype. Unfortunately, their treatment takes place in the context of Codasyl-like databases, and therefore uses a number of structures, like segments and records, that do not belong in CM's.

Cameron [15, p. 226] uses the concept of role in the loose sense that any object executing several processes in parallel is thereby executing several roles. This is not the way we use ''role'' here. Jackson [32, p. 71] mentions the concept of role in roughly the sense we use it here, and excludes it from JSD. So if objects of natural kind person can play the role of reader (of the Daily Racket newspaper), then we must model either readers or persons as "natural kind". If a person can play the role of student first and of teacher later, then we must model this in JSD as two objects of two different natural kinds. This means that in JSD we lose information concerning the identity of the person who first is a student and later a teacher.

4.5. Attributes

Just as the classes that an object has at a particular moment are either essential or contingent, the structures that the object has at any moment are either essential or contingent. An object has a structure *essentially* if it cannot exist without having that structure, otherwise it has the structure *contingently.*

All essential structure of an object is expressed by means of its object identifier. For example,

the structured identifier $\{p_1, ..., p_n\}$ is an essential structure; it cannot exist and be composed differently than it is. In any state of the UoD, $\{p_1, ..., p_n\}$ has elements $p_1, ..., p_n$, and a similar truth holds for structured identifiers composed in other ways than by the set constructor.

Contingent structures may differ in different states of the UoD, and are represented by attribute values. For example, the set of employees in a department, the age of a person, the contents of an order, etc. are represented by attribute values. This is most naturally formalized by attributes as functions. Thus, we could have

```
emps    : DEPT    -> PERSONS
age     : PERSON  -> NAT
max-age : PERSONS -> NAT,
```

where `DEPT` is the type of department identifiers, `PERSON` the type of person identifiers, `PERSONS` the type of finite sets of person identifiers, and `NAT` the type of natural numbers.

4.6. Specification language and semantic structures

We now summarize the results of this section by showing how the semantic structures discussed so far can be specified in CMSL and how these specifications can be given a formal semantics. The version of CMSL we use is defined in [59]. As an example, we give a fragment of a CMSL specification of Jackson's Daily Racket example of [32].

```
object spec PersonAttributes
    import
        PersonIds, Names, Addresses
    identifier sorts
        PERSON
    attributes
        name          : PERSON -> NAME
        address       : PERSON -> ADDRESS
end spec PersonAttributes

object spec ReaderAttributes
    import
        Persons, Booleans, SetsOfSubscriptions
    roles
        READER < PERSON
    attributes
        subscriptions : READER -> SUBSCRIPTIONS
end spec Readers
```

Before going into syntactic and semantic details, we represent the semantic structures specified in the example by the graph in figure 2. The top part of figure 2 forms a very simple aggregation hierarchy, in which persons are represented as an aggregation of names and addresses. In general, the aggregation graph is bigger. For example, in the full specification, `ADDRESS` is the type of identifiers of address objects, with attributes like `zip-code` and `city`. The aggregation "hierarchy" may be any directed graph and can contain cycles.

The unlabeled arrow in figure 2 represents the identity function. It says, in effect, that readers form a subclass of persons. The unlabeled arrows and their endpoints jointly form a taxonomic hierarchy, which is a directed graph containing no cycles. The graph of the taxonomic hierarchy may be separated from the aggregation hierarchy to enhance readability.

Finally, roles are always drawn as dashed boxes, to emphasize their ephemeral nature. To increase the information content of the diagram, one can introduce more special edges. For example, the type `SUBSCRIPTION` would be added in some stage, and have a special edge connecting it with `SUBSCRIPTIONS` which represented the special `element-of` relationship with it. We do not

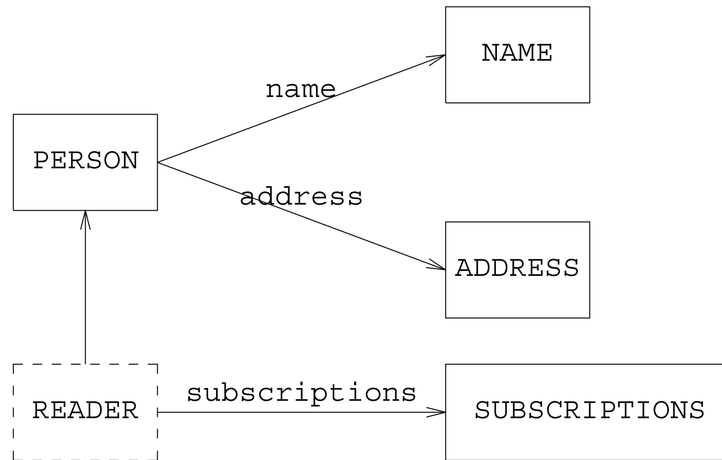

Figure 2.

pursue this further here, because in general there will be semantic information that cannot be represented in a graph but must be represented linguistically (such as the constraint that a reader should have an age above 18). We now briefly summarize some syntactic and semantic points about `PersonAttributes` specification.

In `PersonAttributes`, we import three specifications supposed to have been made elsewhere. `PersonIds` specifies the type `PERSON` of person identifiers

```
p0, p1, p2, p3, ....
```

It also defines a function

```
eq : PERSON x PERSON -> BOOL
```

which tests for equality of two person identifiers. `PersonIds` and `Names` specify the *abstract data types* of person identifiers and letter strings. `Addresses` specifies an *abstract object type,* just as `PersonAttributes` does. Among other things, `Addresses` declares `ADDRESS` to be the type of address identifiers.

In `PersonAttributes`, the type `PERSON` is declared to supply the identifiers for persons. Each person has two attributes, `name` and `address`.

A simple way to think of the formal semantics of `PersonAttributes` is to think of each person as having a set of possible states, such as

⟨p1, ⟨name: "John", address: a15⟩⟩.

We call this a *version* of the person identified by `p1`. In this version of `p1`, the person is called `"John"` and has the address which is identified by `a15`. The set of all versions identified by `p1` is called the *object* identified by `p1`, and the set of all objects identified by identifiers of type `PERSON` is the *class* of person objects. Thus, each person object is identified by a globally unique identifier (principle 6). This class is a *natural kind,* for if an object is a person, it is necessarily (in all states of the UoD) a person (principle 9) and it is subject to all generalizations applicable to persons (principle 8). The only generalization specified in `PersonAttributes` is that all persons have a `name` and `address` attribute. CMSL admits the specification of local and global integrity constraints as well, which limit the allowable versions of a single object or a set of objects, respectively. These constraints are also generalizations that are true of all instances of a class.

`Readers` declares the role `READER` for persons and declares the set of subscriptions held by the reader. Each subscription is assumed to be modeled as an object with an identifier of type `SUBSCRIPTION`, and `SUBSCRIPTIONS` is the type of finite sets of subscription identifiers. The notation

```
READER < PERSON
```

means that the class of readers is at all times contained in the class of persons, but it allows for states in which there is an empty class of readers. `READER` is a role, and the name is introduced in the **roles** section of an object specification.

The formal semantics of the `Readers` specification can be viewed as an extension of the `PersonAttributes` semantics. In addition to the class `PERSON`, we have the class `READER` with elements like

```
⟨p1, ⟨name: "John", address: a15, subscriptions:
{s1, ..., sn}⟩⟩,
```

where `s1`, ..., `sn` are subscription identifiers.

As said before, we will not go into the subject of existence and existence constraints, because this tends to be quite complex. Suffice it to say that there is a mechanism for representing existence in CMSL, and for expressing existence constraints.

4.7. Tasks and heuristics in the object classification step

Table 1 lists the tasks discernible in the object classification step, independently from the conceptual modeling language used.

Task	Heuristics
1. Find the natural kinds of object identifiers.	Decisions to be made are the distinction of objects from masses (principle 5), finding the objects that themselves have no instances (principle 7), and distinguishing natural kinds from roles (principles 8 and 9).
2. Find the roles that objects can play.	Decisions to be made are whether an object can exist without playing a role (principle 10) and which kind of object can play that role (principle 11).
3. Find the attributes of natural kind- and role instances.	No heuristics have been given for this.
4. Find structural relations between object identifiers.	No heuristics have been given for this.
5. Find integrity constraints for natural kinds and roles.	Not treated in this paper.

Table 1. Tasks and heuristics for the object classification step.

Task 5 includes finding the taxonomic relations between the natural kinds and roles discovered in tasks 1 and 2. The *order* of tasks given in the table is not necessarily the order in which they should be executed. It is hard to give any other heuristic on this than ‘‘start with the easiest, continue to the complex, and finish when ready’’. Which task in the object classification step is actually the easiest in a given UoD for a particular analyst, and therefore the best one to start with, depends upon the UoD as well as the analyst (and possibly on the people he or she talks with as well). We return to task ordering briefly in section 7.

5. Event specification

The classification of the UoD in natural kinds, the internal structure of object identifiers, the attributes defined for natural kinds, the constraints defined for natural kinds, and the taxonomic structure of natural kinds and roles are not subject to change. On the other hand, the classification of the UoD into roles currently played by an object, and the attribute values it currently has, depend on the state of the UoD and are subject to change. We need events to model these changes. In general, there are two kinds of events, change of role (including starting or stopping to play a role) and change of

attribute value. We will specify events formally, using results from research in process algebra [8, 9, 10, 46, 47]. In all these algebras, as well as in JSD, we abstract from the *initiative* of an event. Thus, when we say an object executes an event, we mean that it performs or suffers the event. For example, depositing an amount of money may be modeled as an event in the life of a bank account, even though it does not take the initiative to the event. The concept of initiative has not yet been formalized in process theory, but initial studies [44, 61] show that there are interesting connection with concepts like invisible actions in process algebra and responsibility in deontic logic.

Abstracting from initiative, there are two important conceptual issues to be resolved, the localization of an event in conceptual ‘‘space’’ filled by all possible objects and the atomicity of an event in time.

5.1. Locality and the frame assumption

Events are executed by objects, and objects are identified by identifiers. We must therefore localize an event execution by tying it to an object identifier. Because events are coupled to state changes, we will first elaborate a bit on the state of an object.

The concept of a state has been extensively studied in the references on process algebra mentioned above, and turns out to be closely related to the concept of an observation. By an observation of object o_1 by object o_2 we mean an *interaction* between the two objects. We abstract from the asymmetry in observations and consider it to be a symmetric event, i.e. each observation is mutual.

Now, any property that an object currently has, which makes no difference at all for any current or possible future observation of the object, is apparently not relevant for its current or future observable behavior. If it were relevant, then it would make a difference for current or future possible observations. The concept of state in process algebra is based on this idea. The state of an object is, roughly, the set of all possible current and future observations that can be made of the object. So to ask which state an object is in, is equivalent, in a sense formalized in process algebra, to asking which observations can be made of the object in the future by any other object.

Formal details of this can be found in the literature listed above, especially [46]. Here, we assume that the role(s) an object is playing are observable, and its attribute values as well. This means that these are part of the state of an object. It does not mean that the attribute values and role(s) of an object are visible to all objects. But it does mean that there is at least one object that could possibly observe these things. The user of a database system is usually one such object, and there may be more, that reside in the database system.

Now, when we observe an object, we observe an object identified by a unique identifier. Since states are defined in terms of observations, the first general principle to be gleaned from this is that *states are local:*

the state of an object is tied to an object identifier.

Principle 12.

An event may change the state of an object, but it should go without saying that it cannot change the identity of the object executing it. This was already formulated in principle 4, where it is stated that an identifier persists unchanged through all changes. It is because of this principle that we can talk of ‘‘the object executing the event’’.

If an event changes the state of an object, it may do so nondeterministically. That is, execution of the event may lead to any one out of a set of possible next states. We limit nondeterminism to the extent that we require events to be *functions* on the attribute values of an object, leaving nondeterminism to processes in section 6 below. So when event e is executed by an object with attribute a, then after this execution a will have a definite value, and not a set of possible values. So

each event is a function on attribute values.

Principle 13.

There is a connection between state changes and events, which runs as follows. Obviously, when an

object changes its state, it executes an event. (We abstract from the initiative of the event.) The converse is not necessarily true: when an object executes an event, then it may remain in the same state. However, if the event executed by the object is observed by another object, then the observer will change its state, so that we can still say that *something* happened. In general, each event occurrence will cause a state change *somewhere* in the universe, either in the object executing the event, or elsewhere in an observer. When there is no state change anywhere in the UoD, then we will say that there is no event occurrence anywhere in the UoD.

Because states are local (principle 12), state changes are local. Because any state change is an event execution, this means that event executions are local.

An event execution by a single object is local.

Principle 14.

This parallels locality of object state.

Saying that a state is localized by object identifier *p* implies nothing about the state of other objects in the universe. Similarly, saying that *p* executes *e* and thereby changes its state, says nothing about the rest of the universe. However, to determine the effect of an event, we must convert this in a positive statement, viz. that execution of a local event implies that there is no change in the rest of the universe. This consequence is so important that we state it as a separate principle, called the *global frame assumption.*

A local event changes no state in the universe except the local state of the object executing the event.

Principle 15.

Having tied event executions to objects, we can tie the events themselves to object classes. For example, if `inc-age(p)` is executed by person `p`, then `inc-age` is declared to be applicable to all possible persons. By doing this, we assume a great measure of regularity in the universe. Theoretically, it is possible that each object has a repertoire of possible events that differs from the repertoire of events of any other object, including other instances of the same class. If we look around us we see that this is, in fact, not the case and this justifies the the assumption that that

each local event is declared for a natural class and can be executed by all instances of the class for which it is declared.

Principle 16.

This is again an example of how to use classification to describe the infinite by finite means.

5.2. Atomicity of events

An important heuristic given by JSD for finding events is that they must occur in the UoD, not in the system to be implemented [32, p. 65]. This follows from our Principle 1. The only other heuristic given in JSD is that an event is *atomic,* by which two things are meant,

1. events do not persist over a period of time and
2. events are not composed of subevents.

Atomicity of events with respect to time is a feature shared with process algebra. If a duration should be modeled, we should model the start and end of the duration as two distinct, atomic events. Atomicity with respect to other events however, cannot be maintained, for global events will be modeled below as a composition of local events. We can improve on the notion of atomicity as given by Jackson by noting, with Borgers [13], that what is meant is really that an event is a unit of integrity preservation. At one place, Cameron [15 p. 227-278] does this too. This means that an event is simply a function on the admissible states of the CM, where an admissible state is one that satisfies the integrity constraints. For technical reasons, and because we largely ignore integrity constraints in this paper, we will refer to *possible* CM states rather than admissible CM states. The principle of event

atomicity then becomes that

an event execution has no intermediary states.

Principle 17.

This implies that an event is atomic in time, as said before, and that even if an event is composed of subevents, this composite event should still have no intermediary states. In the next section, we discuss *global events*, which are composed of different local events performed or suffered by different objects, and which realize communication between objects. Even though a global event consists of subevents, it still is atomic in the above sense.

5.3. Communication

Communication between objects is modeled in JSD as the execution of common actions by two objects. For example, in Jackson's Widget warehouse example [32], placing an order is an event shared by a customer and an order object, and this is indicated in JSD by specifying the `place` event to occur in the life of an `ORDER` as well as in the life of a `CUSTOMER`. In other words, communication in JSD is *synchronous* and is specified by *event-sharing,* as in CSP [28]. We look at each of these two points in turn.

First, synchronous communication is the joint execution of an event by two or more processes. The JSD representation of communication is inconsistent in this respect, for it requires a *datastream* between communicating processes, as illustrated in figure 3.

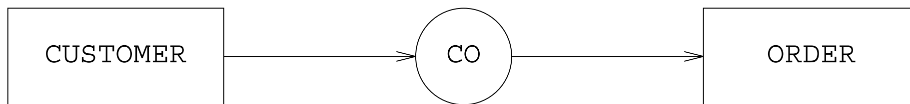

Figure 3.

A datastream is a buffer connected to two processes, a reader and a writer, and transmits the data it receives from the writer to the reader in the order it has received them. In other words, it represents *asynchronous* communication. This not only contradicts synchronicity, it also runs counter to the idea of communication through shared atomic actions. All events, including communication events, should be atomic. When the processes participating in a communication execute their part of the communication at different times, then the system may be in an inconsistent state when part of the communication is executed, but not all of it [cf. 13]. We will therefore model the communication network in a different way, that respects synchronicity, by modeling datastreams as explicit objects, which communicate with other processes synchronously.

Before we do that, we look at the specification of communication. Instead of specifying this by *event-sharing,* as is done in JSD, we will specify it by *encapsulation* and a *communication function.* JSD follows CSP [28] in using event-sharing, and we follow ACP [5, 8, 9, 10] instead, which itself is a generalization of the way communication is specified in CCS [46, 47]. Briefly, the idea is that if event e_1 must communicate with e_2, then we specify that

$$e_1 \mid e_2 = e_3,$$

meaning that e_3 is an event consisting of the synchronous execution of e_1 and e_2. e_3 is still atomic, because it has no intermediate states. $\mid$ is called the communication function in ACP.

Next to specifying the communication function, we encapsulate e_1 and e_2 by renaming them to the deadlock event. This means that any attempt to execute either of them without the other will deadlock. (''Encapsulation'' here means ''shielding from the environment, preventing it from occurring except as part of a communication.'' This differs from the usual meaning of ''encapsulation'' in object-oriented specification, where it means ''locality of states and events.'')

This way of specifying communication has the advantage over CSP that we can be selective in the communication partners, and it has the advantage over CCS that we can specify communication

between events that are not all encapsulated. In CSP, any set of processes that share the event *e* must synchronize on that event, and this may not always be what we want. In CCS, all participating events in a communication are encapsulated, and this may be too strong a demand as well. For example, an `inc-salary` event executed by an employee may or may not synchronize with a `inc-rank` event executed by the same employee, but the `inc-rank` event, if it occurs, must synchronize with `inc-salary`. In that case, `inc-rank` is encapsulated but `inc-salary` is not, and we have

```
inc-rank | inc-salary = promote.
```

We will call encapsulated events *messages.* Both encapsulation and communication take place at two levels, local (in the process executed by a single object) and global (in the process executed by the CM). A *local message* is an event that cannot occur, except when executed synchronously with another event by the same object. A *global message* is an event that cannot occur, except when executed synchronously with another event by a *different* object. Similarly, a communication is called *local* if it consists of events all executed by the same object, and it is called *global* if it consists of events all executed by different objects. A communication is either local or global, but we allow global communications in which participating events are themselves local communications.

We assume the following three principles for communications. First, we assume that events executed by a single object cannot synchronize, except when explicitly specified:

by default, the events executed by a single object are mutually exclusive.

Principle 18.

This means that all local communications must be explicitly specified, and that any local synchronization not specified cannot occur. Across different objects, this principle is reversed:

By default, any set of non-message events executed by a different objects can synchronize.

Principle 19.

This means that only global communications for global messages need to be specified. All other synchronous occurrence can occur anyway, by principle 19. Finally, we assume that the effect of a communication on the object(s) executing it, is a direct sum of the effects of its participating events.

The effect of a communication, executed by different objects, on the object(s) executing it is equal to the effect of the participating events on those object(s).

Principle 19.

For a global communication, this combines neatly with the modularity of objects, because the events executed by the different objects are local and therefore do not interfere. In the case of a local communication, we must be more careful, because the events participating in the communication may very well specify mutually inconsistent updates to the same attributes.

5.4. Specification language and semantic structures

We can extend the example of section 4.6 with events as follows. An informal explanation follows the example.

```
object spec PersonEvents
    import
        PersonAttributes
    events
        change-address : PERSON x ADDRESS -> PERSON
    variables
        p : PERSON
        a : ADDRESS
```

```
      local event constraints
[LEC1]   address(change-address(p, a)) = a
end spec  PersonEvents

object spec  ReaderEvents
      import
          ReaderAttributes
      events
          subscribe : PERSON x SUBSCRIPTION -> READER  global message
          superscribe : READER x SUBSCRIPTION -> READER  global message
          enter      : READER x SUBSCRIPTION -> READER  global message
          win        : READER                -> READER  global message
      variables
          p : PERSON
          s : SUBSCRIPTION
      local event constraints
[LEC1]   subscriptions(subscribe(p, s)) = {s}
[LEC2]   subscriptions(superscribe(p, s)) = subscriptions(p) + {s}
end spec  ReaderEvents

conceptual model spec  DailyRacket
      import
         ReaderEvents, Panels, Mailboxes
      variables
         pp : PANEL
         p  : PERSON
         s  : SUBSCRIPTION
         m  : MAILBOX
      global communications
[GC1]   create-SUBSCRIPTION(db) | subscribe(p, s)
[GC2]   create-SUBSCRIPTION(db) | superscribe(p, s)
[GC3]   award(pp, s) | win(p)
[GC4]   enter(p, s) | receive(m, s)
end spec  DailyRacket
```

`PersonEvents` declares one event for persons, whose effect on the attribute values of the person executing it is constrained by equation `[LEC1]`. The event constraints must be supplemented by the *local frame assumption* that attributes about which nothing can be inferred are not changed by the event. In the formal semantics, we view `change-address(p, a)` as a function on person versions with the following effect:

$$\langle \texttt{p}, \langle \texttt{name: n, address: a'} \rangle\rangle \mapsto \langle \texttt{p}, \langle \texttt{name: n, address: a} \rangle\rangle.$$

In `ReaderEvents`, we see that each person can subscribe to the Daily Racket, and thereby starts playing the role of `READER`. A person may enter as many subscriptions as he or she wants, so there is also a reader event we call `superscribe`, that increases the set of subscriptions the readers already has. Each reader may submit an entry for a competition, which he or she can win. Entry is per subscription, so readers with multiple subscriptions may submit multiple entries, one per subscription.

In the conceptual model specification `DailyRacket`, we import the relevant object specifications and add global information, such as global communication specification. In the example, we do not care to give a name to the specified global communications. To give a name to the global communication `[GC4]`, we would add the event declaration

```
send : READER x MAILBOX x SUBSCRIPTION -> READER x MAILBOX
```

and replace `[GC4]` by

```
[GC4'] enter(p, s) | receive(m, s) = send(p, m, s)
```

The communication structure of the CM can be represented by a *communication diagram,* as illustrated in figure 4.

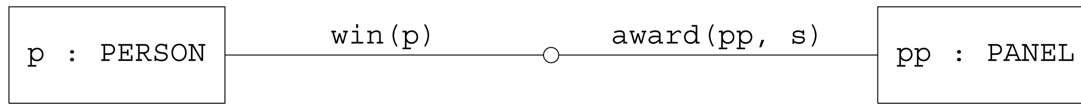

Figure 4.

A communication diagram is a labeled undirected graph with two kinds of nodes, circles and boxes. The boxes in a communication diagram represent particular objects of a natural kind or playing a role. The circles represent *n*-ary communication events, where *n* is the number of incident edges on the circle as well as the number of participating events. Each edge is labeled by the name of the event participating in the communication. If the communication has a name, we can expand the circle and write the name in it.

Note the difference of communication diagrams with datastream diagrams. Datastream diagrams are directed graphs, and the circles represent buffers, not events. By contrast, communication diagrams are undirected graphs, and the circles represent communication events. If the communication represented in figure 4 should be asynchronous, then a mailbox object should be introduced with which persons and panels communicate synchronously.

5.5. Tasks and heuristics

Most of the principles for modeling events have been built into CMSL, so that it is impossible to violate them when specifying CM's in CMSL. Others can be useful as heuristics for finding a model, as indicated in the following table. There are some heuristics we have not discussed. First, natural classes form a taxonomic structure, in which attributes as well as events are inherited from superclasses to subclasses. We do not discuss taxonomies and inheritance in this paper. (They are treated in detail in [59].) Suffice it to say that attributes and events are both required to have globally unique names, and are inherited by all sub–classes of a class. There is one exception to this, *role changes* are not inherited at all. If *e* is the event in which a person starts playing the role of a student, then it makes no sense to make this event applicable to all subclasses of persons. For example, the class of students is a subclass of the class of persons, and does not inherit *e*. At the moment, we have not yet been able to formulate a general rule for role-change inheritance that cannot be violated by a valid counterexample. We therefore provisionally use the rule that role changes are not inherited at all.

Secondly, having formally specified communications, we may be able to prove freedom of deadlock for particular specifications. For examples of such proofs, we refer to the literature on process algebra [e.g. 4]. This can lead to improvements of the process specification. Deadlock elimination is the formal pendant of the elimination of ordering clashes in JSD, in which a conflict in the sequence of events in communicating processes is eliminated [cf.32, p. 231].

Thirdly, event preconditions are not discussed in this paper. These are sets of equations which specify when an event can be executed. If the equations are false, then the event cannot be executed. They may be defined for local as well as global events, and can be used to preserve local and global integrity constraints. CMSL provides a logic for proving constraint invariance under event preconditions.

Jackson [32] and Sutcliffe [54] give a number of more pragmatic heuristics to discover actions. One may for example look for verbs (‘‘borrow’’, ‘‘cancel’’) and nouns (‘‘cancellation’’, ‘‘delivery’’) in UoD descriptions. Synonyms must be reduced to one verb or noun, and system output events, or events required to execute a function of the system, must be eliminated again. All of this follows from principle 1 and the language we use. Since the language we use is a contingent matter, we did

Task	Heuristics
1. Define local events for each natural class.	Events take place in the UoD (principle 1), are atomic (principle 17), take place when the local state of an object is changed (principles 12, 15), and are declared for all instances of a natural class (principle 16).
2. Define role changes.	Each role must be playable. There must thus be an event which causes an object to play a role.
3. Define the effect of each event on the state of its subject.	A local event is a function on the possible states of its object (principle 13). The effect of a communication event is equal to that of its participating events (principle 20).
4. Define the messages.	A local message cannot occur, except when synchronized with another event executed by the same object. A global message cannot occur, except when synchronized with another event executed by a different object.
5. Specify communications.	A local communication is equal to a set of local events executed by the same object, a global communication is equal to a set of local events, each executed by a different object.
6. Check for possibility of deadlock.	Not treated in this paper.
7. Specify event preconditions.	Not treated in this paper.

Table 2. Tasks and heuristics for the event specification step.

not bother to elevate considerations related to it to heuristic principles. (In Chinese, for example, the distinction between nouns and verbs is not so clear-cut as in English. But information systems *are* developed in China.)

6. Process specification

So far, we have specified objects and their attributes, and the events which may change the attribute values of an object. We now come to the heartland of JSD, the specification of the life-cycle an object goes through from birth to death. We immediately go to the specification and semantic structures of object life cycles, and formulate principles afterwards.

6.1. Specification and semantic structures

To specify the life cycle of the persons and readers of our example in CMSL, we extend `ReaderEvents` as follows.

```
object spec PersonAndReaderProcess
     import
          ReaderEvents, ProcessAlgebra
     process
[P1]      PERSON = CHANGE-ADDRESS || subscribe
[P2]      CHANGE-ADDRESS = change-address . CHANGE-ADDRESS
[P3]      READER = CHANGE-ADDRESS || ENTER || SUPERSCRIBE || win
[P4]      ENTER = enter . ENTER
[P5]      SUPERSCRIBE = (superscribe . ENTER ) ||_ SUPERSCRIBE
end spec PersonAndReaderProcess
```

This specification imports a specification ProcessAlgebra, that declares the events of persons and readers as constants of type EVENT. Thus, we must customize the ProcessAlgebra specification to the repertoire of events of the objects whose life cycle we want to specify. ProcessAlgebra also specifies operators to combine events into processes. Some of these operators are + for choice, . for sequential composition, and || for parallel composition. EVENT is a subtype of PROCESS, and all operators are defined for processes and yield processes. So

```
change-address + subscribe
```

is the process which has the choice between change-address and subscribe, and in

```
(change-address + subscribe) || win
```

this is executed in parallel with the win event.

The specification ProcessAlgebra has a model, which consists of all processes satisfying the axioms. We use the *graph model,* in which each process can be represented by a rooted directed graph. We refer to the work by Bergstra & Klop for a precise definition of the process operators and the graph model [5, 8, 9, 10].

The equations in the process section specify a process for each natural class. The constants in these equations are events written in lower-case letters. The variables range over processes and are written in upper-case letters. The variables need not be declared, for they are all of type PROCESS. For each natural class, there should be a variable with the same name as the class, called a *main variable* of the process specification. In the example, there are actually two process specifications, [P1-2] with main variable PERSON, and [P2-5] with main variable READER. The process equations are solved for the main variables in the process model and should be *guarded,* which means roughly that the equations can be written in such a form that each variable in a sequence is preceded by a constant. It can be shown that the equations then have precisely one solution.

[P1-2] describe the life of a person as a simple iteration of address changes in parallel with one execution of subscribe. One effect of the last event is that the person becomes a reader, and so starts executing a life as a reader. This is not visible in the process specification, because we adhere to the principle that the process executed by instances of a more specialized class, like READER, should be invisible when we look at the process executed by instances of a more general class, like PERSON. However, the declaration of subscribe shows that the person executing it starts playing the role of a reader. Figure 5 illustrates the person process.

PERSON

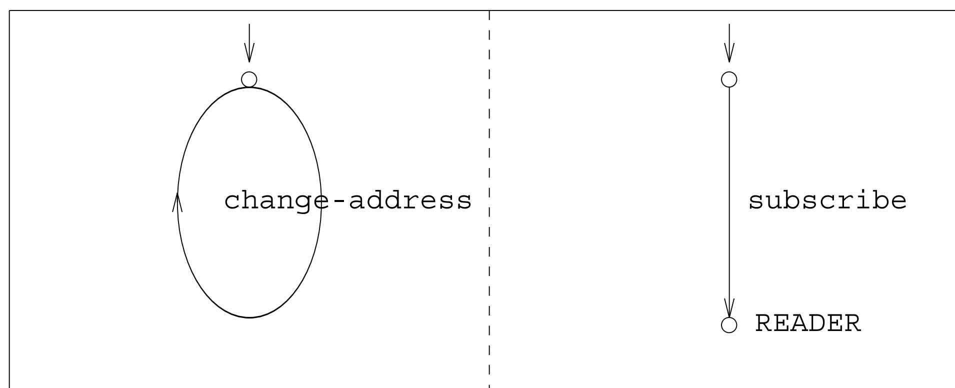

Figure 5.

A parallel composition of process graphs is roughly the Cartesian product of the graphs, and we use Harel's higraphs to represent this [26]. The graphs of the Cartesian product are displayed in a box separated by a dashed line. Each graph is a rooted directed graph, with the root indicated by a small arrow. We show the effect of subscribing on the role a person is playing next to the result state of subscribe.

Figure 6 shows the higraph for the reader process. A reader executes a parallel composition of an iteration over `change-address`, `enter`ing a competition, adding more subscriptions to his or her collection, and winning. A reader may enter a competition for each subscription he or she has. This constraint is not expressed in the process specification, but must be specified as a precondition on the `enter` events. A hidden rule of the game is that each reader can win only once, regardless how many subscriptions he or she has. This is expressed in the single occurrence of the `win` event in the `READER` process.

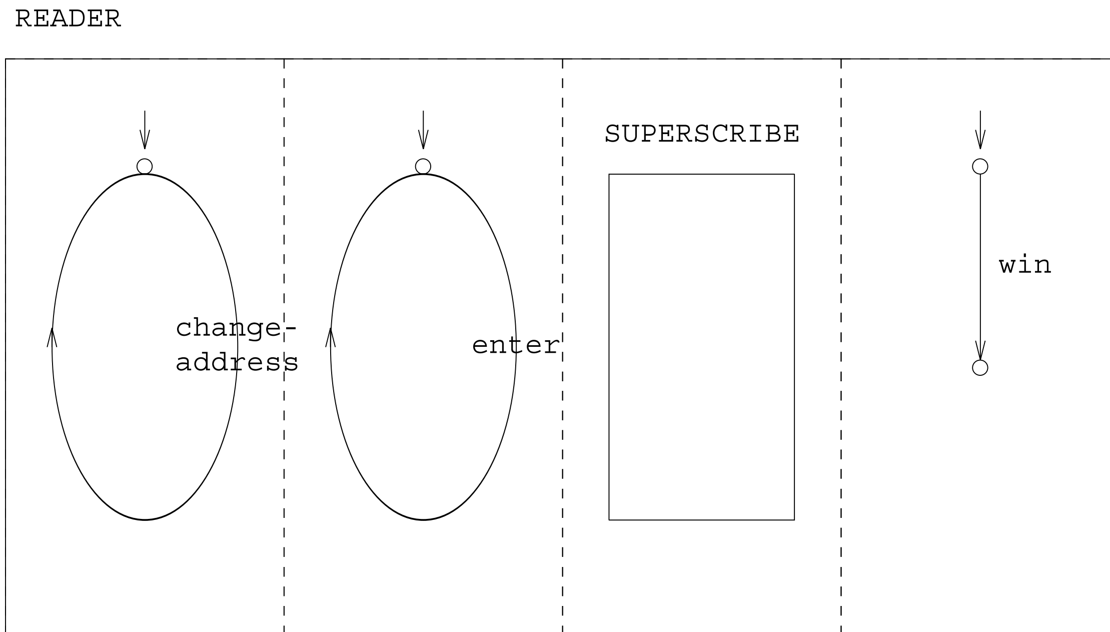

Figure 6.

The binary operator `||_` in `[P5]` is called *left-merge,* and is a parallel composition in which the first event is required to be executed by the left-hand argument. We show this in the higraph by extending the arrow at the start node so that it enters the box from the outside (figure 7).

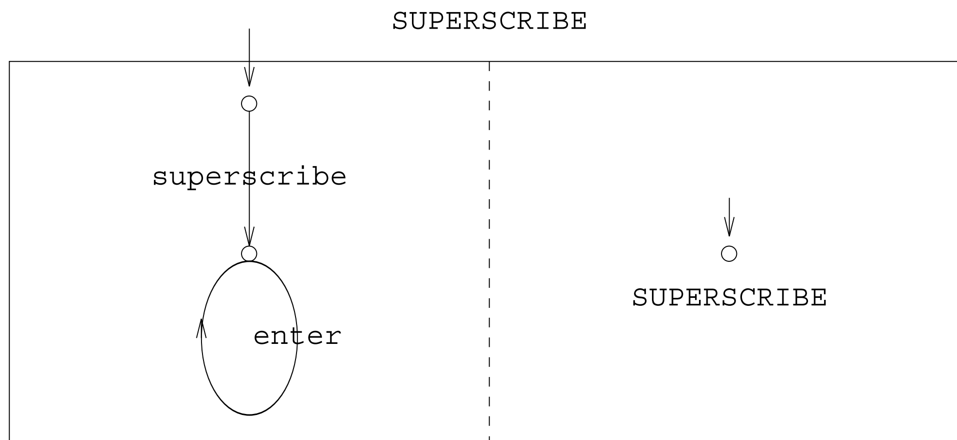

Figure 7.

Figure 7 also shows a recursive higraph: the initial state of the right-hand box is the complete `SUPERSCRIBE` box. A state in a higraph can be identified with a box shown elsewhere in the diagram. Thus, figure 7 shows that, in order to execute the process in the right-hand side box, we execute the start event in the `SUPERSCRIBE` box. To sum up, `[P5]` expresses the fact that after each subscription, a reader has the right to start an `ENTER` process, and may continue to gain more subscriptions.

Note that the reader process includes the person process except for the role change event `subscribe`. In general, we require the life cycle of a subclass to include the life cycle of each of its superclasses. This is possible, because events declared for a class are inherited by all subclasses. The only exception on this in CMSL is that role changes are not inherited (see section 5.5). So `subscribe` does not re-appear in the `READER` process.

6.2. Tasks and heuristics

The first important principle in process specification is that, just like for events,

each process is defined for a natural class.

Principle 21.

This is analogous to principle 16 for events. In CMSL, we adopt the convention that if no process is specified for a class *x*, then it simply inherits the processes of its superclasses, to which is added an unstructured iteration over the events declared specifically for *c*. We briefly return to the topic of process inheritance below.

Not just any process observed in the UoD can be elevated to the life cycle of an object. A process specification for instances of a natural class is a universal statement about these instances, which holds for these instances without exception. We can classify universal statements in two ways [43, 57, 58]. One way is to divide them into *static* and *dynamic* statements. Universal statements of the static kind state something about the set of possible states of the UoD, while those of the dynamic kind state something about the state changes of the UoD. The process specification tied to a natural class is a universal statement of the dynamic kind. The other way to classify universal statements is orthogonal to this, and divides them into *analytical, empirical,* and *normative* statements. Analytically true statements are true because of the meaning of the words occurring in it, and empirically true statements are true because the world happens to behave in the way specified by the statement. Normative statements are not true or false, they are obeyed or violated by the world.

It turns out that in the UoD's of most information systems, there are hardly any dynamic universal statements that are also empirical. The process specifications are either analytically true or they are prescriptions for the behavior of objects in the UoD. For example, it is an analytical truth that an employee cannot be fired before she is hired. This could result in the specification the process `hire . fire` for employees. This process is followed by all possible employees, without exception, because of the meaning of ''hire'' and ''fire''. As an example of a normative universal statement, a book borrowed from a library must be returned afterwards, without exception. This motivates a process like `borrow . return` for library members. In addition to eliminating the logical impossibility that a book is returned before it is borrowed, it enforces the behavior that a book, as a matter of fact, is returned after being borrowed.

The criterion for allowing these processes to be attached to a natural class is that all instances of the class, without exception, follow these processes. Observations like ''most of the time, students follow course A before they follow course B'' may be empirically valid, but are no good as process specifications form a natural class. To be elevated to that status, the statement must have the force of a necessary truth, like ''all students follow A before they do B''.

Most of the time, only a weaker kind of statement can be made, such as ''all students *must* follow A before they do B''. These statements are of a normative nature. If we want to model these as object life cycles, then they must be norms that allow no exceptions, i.e. they must be *totalitarian norms.* In general, processes follow an inexorable logic which either follows from the logic of the terms used in it or follows from the totalitarian nature of the norms it expresses.

The process executed by instances of a natural class allows no exceptions.

Principle 22.

This principle is also stated in [32, pp. 100 ff.], [33], and [54, p. 37]. For example, of any event ordering not allowed by the process specification, we should be able to say ''it doesn't happen''.

An important concept in JSD life-cycle modeling is that of a *marsupial* entity [32, pp. 100 ff.], [54, p. 37]. The adjective ''marsupial'' does not indicate anything special about the entity itself, but about the way we discover its existence: we discover it as part of another entity which should be modeled as an independent entity. There are two elements in this. First, we may discover that an object executes two or more processes in parallel. For example, a car may execute a process containing changes in its physical nature (painting, tuning, repairing) in parallel with a process containing changes in its legal status (buying, selling). Neither of these processes constitutes a marsupial in the JSD sense, for they are both processes executed by a single object. Second, we may discover that one of the processes executed by the object is really a role played by the object, or is really a process executed by a different object. For example, the buying and selling process of a car may be defined as the process belonging to the role PROPERTY, playable by cars and possibly by other kinds of objects as well. Or we may discover that the customer process we specified is really an account process, where one customer can open many accounts. In JSD, both PROPERTY and ACCOUNT will be modeled as marsupials, without taking the difference between roles and natural kinds into account.

The principle that processes be without exceptions can be construed as saying that all exceptions should be stated as part of the process specification. In practice, this requires distinguishing a normal course of life and an abnormal course of life of a natural class of objects. Each abnormal course can be viewed as a trace of 0 or more events through the process representing the normal course of life, interrupted by an event introducing the abnormal course, followed by a wind-up process dealing with the abnormal case. This may end the life of the object, or it may return to resume the normal course of life. In JSD, only the first case is considered, which is called *premature end* [32, p. 107]. It is modeled using a kind of interrupt processing technique: we assume that the normal course of events prevails, until it turns out that the contrary is the case. Then we jump to an appropriate interrupt-handling process, where we deal with the abnormal situation. There is no special construct for this in CMSL, since it can dealt with using the current version of the language. However, to structure the process specification, it may be desirable to add a facility like Bergstra's mode transfer operator [11] later on in the development of the language.

Finally, for all instances of natural kinds, we must be sure that they can be created and/or deleted, and for all roles we must be sure that they can be played and/or that objects can stop playing these roles. In JSD terms, we must establish *entity boundaries.* We will call events that create or delete an object or that cause an object to start or stop playing a role *boundary events.*

The tasks and heuristics in the process specification step are shown in table 3. Task 5, checking the taxonomic structure of processes, is not treated here. Suffice it to note that there is still not a satisfactory formalization of the concept that the process of a subclass should somehow "contain", be more specific as, the processes specified for all of its superclasses. Discussions of this can be found in [20, 59].

7. A note about task ordering

Tables 1-3 summarize the tasks we identified in the modeling process of dynamic objects. They have been grouped into three sets, object classification, events specification, and process specification, and within each set, the tasks have been numbered. This does not imply that they should be executed in this order. For example, it may seem common sense to specify events before we specify processes, because processes are built from events. However, in many cases, the processes will be business policies that are formulated in natural language, and are gradually explicated until they are specified formally. It is quite conceivable that only after this, we bother to specify the effect of the events on attribute values. Again, we may want to start specifying the communication patterns of an organization before thinking about other events or about the attributes of objects. To give a sensible advice about task ordering, we must first gain more experience with object-oriented modeling. In the absence of this advice, we can just as well start with task 1 of the object classification step, unless there is some compelling reason to start elsewhere.

Task	Heuristics
1. Compose events into process.	For each natural class a process is specified (principle 20).
2. Look for marsupials.	Where there are marsupials, there is parallelism. This may indicate parallel processing by a single object o, or a role executed by a single object, or a process executed by a different object, mistakenly identified with o.
3. Check for exceptions.	All processes should allow no exceptions. Most of the time, the processes follow from the meaning of the words used to describe the UoD, or they are totalitarian prescriptions for behavior of objects. All possible exceptions should be explicitly specified.
4. Check for boundary events.	Each natural kind should have creation and/or deletion events, and each role should have a start and/or stop event.
5. Check the taxonomic structure of processes.	Not treated in this paper.

Table 3. Tasks and heuristics for the process specification step.

8. Summary and conclusions

We defined a method as a combination of a specification language, CM structures, and pragmatic advice about the tasks to be executed to specify a CM of a UoD. The specification language used in this paper is CMSL, described in more detail elsewhere [59, 60, 62, 65]. We concentrated on the task of finding an object-oriented CM and divided it into three subtasks, object classification, event specification, and process specification. Each of these tasks have been divided into a number of subtasks, for which heuristics have been given.

Most of the tasks and heuristics have been identified by a logical analysis of object-oriented CM structures, while some have been translated from JSD into object-oriented terms. What has been achieved in this way is a first, motivated version of an object-oriented analysis method which can yield formally specified object-oriented CM's. This is a beginning of an answer to the question how object-oriented models should be developed. To continue developing the answer, we should both expand the method as proposed here, and test it on real-life cases. By applying it to real-life cases, we hope to identify the weak spots in the current version of CMSL, as well as to find more heuristics for the tasks we identified, and find suggestions for ordering the tasks in particular cases.

To expand the method, we will look at other methods known to be practical, like Structured Analysis [19], SADT [41], SSADM [1], and NIAM [48]. Some of this has already been done [63, 66], other work is still in progress. In addition, we will look at novel ways new kinds of integrity constraints, such as the normative constraints studied in [42, 58, 67].

Acknowledgements: CMSL was defined as part of Ph.D. research done at the Vrije Universiteit, Amsterdam, under the supervision of Reind van de Riet. My understanding of process algebra benefited from discussions with Prof. Jan Willem Klop. Much of the work on formal aspects of conceptual modeling has been done in cooperation with Prof. John-Jules Meyer, Frank Dignum and Hans Weigand. Finally, I would like to thank the two anonymous referees who contributed to the paper through constructive criticism.

9. References

1. C. Ashworth and M. Goodland, *SSADM: A Practical Approach*, 1990.
2. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, ''The Object-Oriented Database System Manifesto,'' pp. 40-57 in *Proceedings, The First International Conference on Deductive and Object-Oriented Databases*, ed. W. Kim J.-M. Nicolas S. Nishio, Kyoto (december 4-6, 1989).
3. C.W. Bachman and M. Daya, ''The Role Concept in Data Models,'' pp. 464-476 in *Proceedings of the Third International Conference on Very Large Databases* (1977).
4. J.C.M. Baeten and J.A. Bergstra, ''Global Renaming Operators in Concrete Process Algebra,'' *Information and Control*, pp. 205-245 (1988).
5. J.C.M. Baeten and W.P. Weijland, *Process algebra,* Cambridge University Press (1990). Cambridge Tracts in Theoretical Computer Science 18.
6. J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, and D. Woelk, ''Data Model Issues for Object-Oriented Applications,'' *ACM Transactions on Office Information Systems* **5**, pp. 3-26 (january 1987).
7. C. Beeri, ''Formal Models for Object Oriented Databases,'' pp. 370-395 in *Proceedings, The First International Conference on Deductive and Object-Oriented Databases*, ed. W. Kim, J.-M. Nicolas, S. Nishio &, Kyoto (december 4-6, 1989).
8. J.A. Bergstra and J.W. Klop, ''Process Algebra for Synchronous Communication,'' *Information and Control* **60**, pp. 109-137 (1984).
9. J.A. Bergstra and J.W. Klop, ''Algebra of Communicating Processes with Abstraction,'' *Theoretical Computer Science* **37**, pp. 77-121 (1985).
10. J.A. Bergstra and J.W. Klop, ''Algebra of Communicating Processes,'' pp. 89-138 in *Mathematics and Computer Science (CWI Monographs 1)*, ed. J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, North-Holland (1986).
11. J.A. Bergstra, ''A Mode Transfer Operator in Process Algebra,'' Programming Research Group Report P8808, University of Amsterdam (April 1988).
12. G. Booch, ''Object-Oriented Development,'' *IEEE Transactions on Software Engineering* **SE-12**, pp. 211-221 (1986).
13. M. Borgers, ''Een Bijdrage tot de Verdere Ontwikkeling van de JSD-Methode,'' *Informatie* **30**, pp. 549-565 (1988).
14. J.R. Cameron, *JSP & JSD: The Jackson Approach to Software Development,* Computer Society Press (1983).
15. J.R. Cameron, ''An Overview of JSD,'' *IEEE Transactions on Software Engineering* **SE-12**, pp. 222-240 (1986).
16. B. Carr, *Metaphysics: An Introduction,* MacMillan (1987).
17. P.P.S. Chen, ''The Entity-Relationship Model: Toward a Unified View of Data,'' *ACM Transactions on Database Systems* **1**, pp. 9-36 (1976).
18. E.F. Codd, ''Extending the Database Relational Model to Capture More Meaning,'' *ACM Transactions on Database Systems* **4**, pp. 397-434 (1979).
19. T. DeMarco, *Structured Analysis and System Specification,* Yourdon Press/Prentice-Hall (1978).
20. H.-D. Ehrich, A. Sernadas, and C. Sernadas, ''Objects, Object Types, and Object Identification,'' pp. 142-156 in *Categorical Methods in Computer Science*, ed. H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, Springer (1987). Lecture Notes in Computer Science 393.
21. D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derret, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan, ''Iris: An Object-Oriented Database Management System,'' *ACM Transaction on Office Information*

Systems **5**, pp. 48-69 (January 1986).

22. M.T. Ghiselin, ''Species Concepts, Individuality, and Objectivity,'' *Biology and Philosophy* **2**, pp. 127-143 (1987).
23. J.A. Goguen and J. Meseguer, ''Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics,'' pp. 417-477 in *Research Directions in Object-Oriented Programming*, ed. B. Shriver & P. Wegner, MIT Press (1987).
24. J.A. Goguen and D. Wolfram, *On Types and FOOPS,* To be published, IFIP TC2 Working Conference on Database Semantics, Windermere, U.K. (2-6 july 1990).
25. P. Hall, J. Owlett, and S. Todd, ''Relations and Entities,'' pp. 201-220 in *Modelling in Database Management Systems*, ed. G.M. Nijssen, North-Holland (1976).
26. D. Harel, ''On Visual Formalisms,'' *Communications of the ACM* **31**, pp. 514-530 (1988).
27. W. Hennig, ''Phylogenetic Systematics,'' pp. 603-622 in *Conceptual Issues in Evolutionary Biology*, ed. E. Sober, MIT Press (1984).
28. C.A.R. Hoare, *Communicating Sequential Processes,* Prentice-Hall (1985).
29. S.E. Hudson and R. King, ''CACTIS: A Database System for Specifying Functionally-Defined data,'' pp. 26-37 in *International Workshop on Object-Oriented Database Systems*, IEEE (1986).
30. D.L. Hull, ''The Effect of Essentialism on Taxonomy -Two Thousand Years of Stasis, Part I,'' *The British Journal for the Philosophy of Science* **15**, pp. 314-326 (1965).
31. D.L. Hull, ''The Effect of Essentialism on Taxonomy -Two Thousand Years of Stasis, Part II,'' *The British Journal for the Philosophy of Science* **16**, pp. 1-18 (1965).
32. M. Jackson, *System Development,* Prentice-Hall (1983).
33. JSD Ltd., *JSD Course Notes,* Michael Jackson Limited (1986).
34. W. Kent, *Data and Reality,* North-Holland (1978).
35. S.N. Khoshafian and G.P. Copeland, ''Object Identity,'' *Object-Oriented Programming Systems, Languages and Applications*, pp. 406-416, SIGPLAN Notices 22 (12) (1986).
36. M. Kifer and G. Lausen, ''F-Logic: A Higher-Order Language for reasoning About Objects, Inheritance, and Scheme,'' *Proceedings of the ACM/SIGMOD International Symposium on Management of Data*, Portland, pp. 134-146, SIGMOD RECORD vol. 18, nr. 2 (june 1989).
37. S. Kripke, *Naming and Necessity,* Basil Blackwell (1980). Second edition
38. M.J. Loux, *Substance and Attribute. A Study in Ontology,* Reidel (1978).
39. M.J. (ed.) Loux, *Universals and Particulars: Readings in Ontology,* Doubleday (1970).
40. D. Maier and J. Stein, ''Development and Implementation of an Object-Oriented DBMS,'' pp. 317-354 in *Research Directions in Object-Oriented Programming*, ed. B. Shriver & P. Wegner, MIT Press (1987).
41. D.A. Marca and C.L. Gowan, *SADT: Structured Analysis and Design Technique*, 1988.
42. J.-J.Ch. Meyer, ''A Different Approach to Deontic Logic: Deontic Logic Viewed as a Variant of Dynamic Logic,'' *Notre Dame Journal of Formal Logic* **29**, pp. 109-136 (winter 1988).
43. J.-J. Ch. Meyer, H. Weigand, and R.J. Wieringa, ''A Specification Language for Static, Dynamic and Deontic Integrity Constraints,'' pp. 347-366 in *2nd Symposium on Mathematical Fundamentals of Database Systems*, ed. J. Demetrovics, B. Thalheim, Springer, Visegrád, Hungary (june 1989). Lecture Notes in Computer Science 364.
44. J.-J.Ch. Meyer and R.J. Wieringa, *Actor-Oriented System Specification with Dynamic Logic*, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), April 1991.
45. J.S. Mill, *A System of Logic,* Longman (1970). First ed. 1843
46. R. Milner, *A Calculus of Communicating Systems,* Springer (1980). Lecture Notes in Computer

Science 92.

47. R. Milner, *Communication and Concurrency,* Prentice Hall (1989).

48. G.M. Nijssen and T.A. Halpin, *Conceptual Schema and Relational Database Design,* Prentice-Hall (1989).

49. H. Putnam, ‘‘The Meaning of ’Meaning’,’’ *Mind, Language and Reality*, pp. 215-271, Cambridge University Press, H. Putnam (1975).

50. S.P. Schwartz, ‘‘Natural Kinds and Nominal Kinds,’’ *Mind* **89**, pp. 182-195 (1980).

51. S.P. (ed.) Schwartz, *Naming, Necessity, and Natural Kinds,* Cornell University Press (1977).

52. A. Sernadas, C. Sernadas, and H.-D. Ehrich, ‘‘Object-Oriented Specification of Databases: an Algebraic Approach,’’ pp. 107-116 in *Proceedings of the Thirtheenth International Conference on Very Large Databases*, ed. P.M. Stocker & W. Kent, Brighton (1987).

53. A. Sernadas, J. Fiadeiro, C. Sernadas, and H.-D. Ehrich, ‘‘The Basic Building Blocks of Information Systems,’’ pp. 225-246 in *Information System Concepts: An In-Depth Analysis*, ed. E.D. Falkenberg & P. Lindgreen, North-Holland (1989).

54. A. Sutcliffe, *Jackson System Development,* Prentice-Hall (1988).

55. F.E. Warburton, ‘‘The Purposes of Classifications,’’ *Systematic Zoology* **16**, pp. 241-245 (1967).

56. H. Weigand, *Linguistically Motivated Principles of Knowledge Base Systems,* Foris, Amsterdam (1989). PhD Thesis, Free University

57. R.J. Wieringa, ‘‘Three Roles of Conceptual Models in Information System Design and Use,’’ pp. 31-51 in *Information System Concepts: An In-Depth Analysis*, ed. E.D. Falkenberg & P. Lindgreen, North-Holland (1989).

58. R.J. Wieringa, J.-J. Ch. Meyer, and H. Weigand, ‘‘Specifying Dynamic and Deontic Integrity Constraints,’’ *Data and Knowledge Engineering* **4**, pp. 157-189 (1989).

59. R.J. Wieringa, ‘‘Algebraic Foundations for Dynamic Conceptual Models,’’ Ph.D Thesis, Vrije Universiteit, Amsterdam (May 1990).

60. R.J. Wieringa, *Equational Specification of Dynamic Objects*, Proceedings, IFIP TC2 Working Conference on Database Semantics, 2-6 july 1990.

61. R.J. Wieringa and J.-J.Ch. Meyer, ‘‘Actor-Oriented Specification of Dynamic and Deontic Integrity Constraints,’’ Technical Report, Department of Mathematics and Computer Science, Vrije Universiteit (October 1990). To be published, Third Int. Symp. on Mathematical Foundations of Database and Knowledge Base Systems, May 1991.

62. R.J. Wieringa, ‘‘An Integrated Specification of Values, Objects and Processes for Object-Oriented Models,’’ *Proceedings, 2nd International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria (23-28 september 1990).

63. R.J. Wieringa, *Object-Oriented Analysis, Structured Analysis, and Jackson System Development,* Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (December 1990). To be published, Proceedings of the IFIP TC8/WG8.1 Working Conference on the Object-Oriented Approach to Information Systems, Quebec City, Canada, October 1991.

64. R.J. Wieringa, ‘‘A Conceptual Model Specification Language (CMSL Version 2),’’ Technical Report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (April 1991).

65. R.J. Wieringa, *A formalization of objects using equational dynamic logic,* Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (January 1991). Technical report, submitted for publication.

66. R.J. Wieringa, *Principles of Conceptual Modeling,* Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (1991). Course text.

67. R.J. Wieringa, H. Weigand, J.-J. Ch. Meyer, and F. Dignum, ‘‘The Inheritance of Dynamic and

Deontic Integrity Constraints,'' *Annals of Mathematics and Artificial Intelligence* **3**, pp. 393-428 (1991).

68. Wolterstorff, *On Universals. An Essay in Ontology,* The University of Chicago Press (1970).

69. C. Zaniolo, ''Object Identity and Inheritance in Deductive Databases -an Evolutionary Approach,'' pp. 2-19 in *Proceedings, The First International Conference on Deductive and Object-Oriented Databases*, ed. W. Kim J.-M. Nicolas S. Nishio, Kyoto (december 4-6, 1989).

70. S.B. Zdonik and D. Maier, ''Fundamentals of Object-Oriented Databases,'' pp. 1-32 in *Readings in Object-Oriented Database Systems*, ed. S.B. Zdonik, D. Maier, Morgan Kaufmann (1990).