A conceptual model specification language (CMSL Version 2)

R.J. Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV, Amsterdam
The Netherlands
Email: roelw@cs.vu.nl

June 30, 1992

Abstract

Version 2 of a language (CMSL) to specify conceptual models is defined. Version 1 was defined in may 1990 [22]. CMSL consists of two parts, the value specification language VSL and the object specification language OSL. In chapter 1, an informal explanation of the difference between objects and values is given, after which VSL and OSL are introduced by means of small examples in chapters 2 and 3. There is a formal semantics of CMSL and an inference system for CMSL [29, 23, 24, 22, 26], but research on this still continues. A method for developing CMSL models is also being developed [25, 27]. A start has been made with a workbench for CMSL [11, 17, 18, 19, 20]. This report is a working document to keep track of the current results and research problems of CMSL and is not intended for publication.

Chapter 1

Introduction

1.1 The purpose of CMSL

CMSL (Conceptual model specification language) is an executable specification language with which conceptual models (CM's) can be specified. We assume that the CM's specified by CMSL are used to represent a part of reality that is to be implemented in a database (DB) system. This is illustrated in figure 1.1.

The CM specification has a CM as its *intended model*. This means that there is an agreement among the users of CMSL that among all the possible models of a CMSL specification, there is one which is meant by the specifier, and that it is known among the users of CMSL which of the possible models this is. For example, Datalog specifications have an Herbrand model as their intended model.

The CM in its turn represents a part of reality called the *object system*. Often, the object system is part of a larger piece of relevant reality, called the universe of discourse (UoD), that includes the object system, the DB system implementing a CM, and the users of the DB system. The CM is usually not *equivalent* to the object system, because it will leave out many details. It is an abstract representation of the object system. However, it may be said to contain knowledge about the object system.

By requiring the specification to be *executable*, we require it to be executable by a computer. We can assume that there is a programmable system, usually called a *database management system* (DBMS), into which the specification can be entered, such that the resulting system *simulates* the CM. The resulting system is called a *database system* (DBS). Thus, the DBS is a system that behaves as if it were the CM, but it is not equal to the CM. For

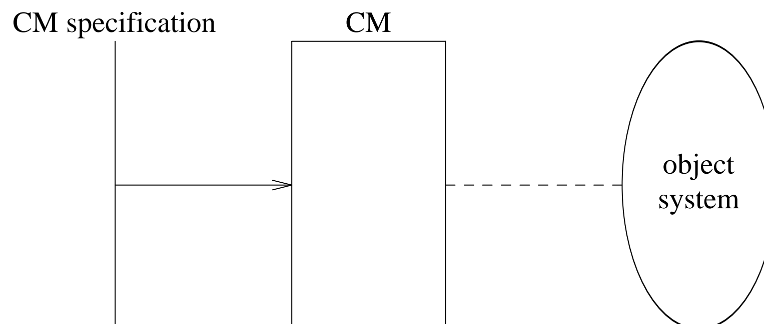

Figure 1.1: A CM, its specification, and reality.

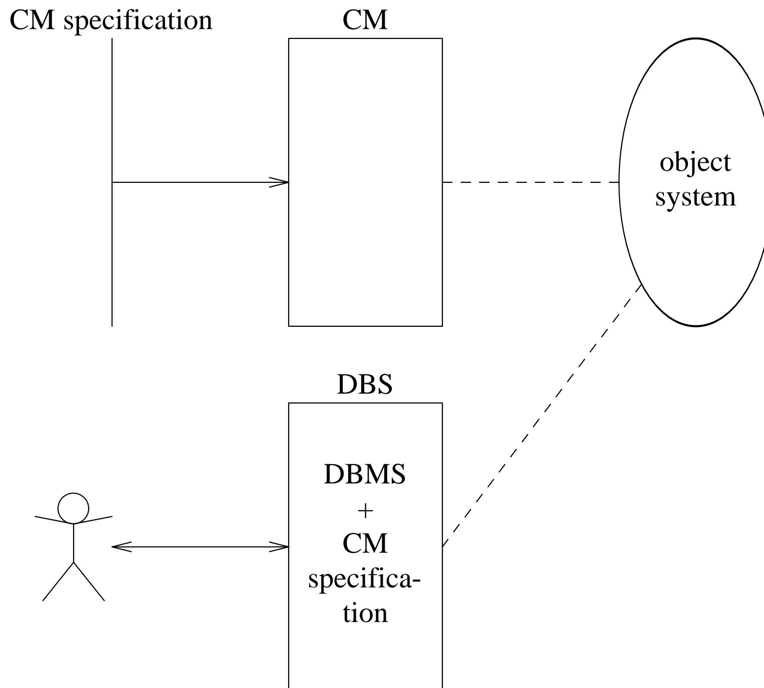

Figure 1.2: A DBS and its CM.

example, a CM may contain all the natural numbers, but a DBS will never store more than a finite set of natural numbers. However, to the user of the DBS, the DBS appears as if it were a CM of the object system. This is illustrated in figure 1.2. The user interacts with a DBMS in which the CM specification is entered, and which behaves as if it were the CM of the object system. Because figure 1.1 represents *what* the DBS must do and figure 1.2 shows how it does this, we will think of CMSL and its role primarily in terms of figure 1.1.

1.2 Objects and values

The models specified by CMSL represent the object system by means of **values** and **objects**, and CMSL therefore has two sublanguages, the *value specification language* VSL and the *object specification language* OSL. These are described in chapters 2 and 3. In this section we describe intuitively what the difference between values and objects is.

A good informal description of the distinction between values and objects is given by MacLennan [14]. The following account is based upon this and on an analysis given in Wieringa [22]. To determine thoughts, take the natural numbers as typical examples of values, and the set of all possible cars as typical examples of objects.

First of all, values and objects both are **discrete entities**, by which we mean that any collection of values and/or objects forms a *set*. Thus, a collection of cars and/or natural numbers is a set. This contrasts with masses like water or gold. A mass of water is not a set, and therefore water is not a discrete entity.

Second, values and objects both have **properties**, by which we mean *possible observations* of an entity. Thus, the weight, color, number of doors, etc. of a car are properties of a car that can be observed. Similarly, we can observe of a natural number that it is a prime number or that it is an odd or even number.

We will not say what an observation is, except that it is an interaction between two entities, an observer and an observed, in which information passes from the observed to the observer.

Third, we include in the possible observations of an entity all possible future and past observations, and all observations that have been possible once or may become possible once. Now, here there is a difference between cars and natural numbers. All possible observations of a natural number could not be otherwise than they are. 3 is an odd prime number once and for all, and no set of possible observations will ever show it to have different properties. Among all possible observations of a car, on the other hand, there are many that could have been different. The color, weight, number of doors etc. of a car could have been different if the builders of the car had made it different. It is open to discussion how many of the properties of the car could have been different without talking about a different car, but at least there are properties, like color, that could have been different for the *same* car.

We call the set of all properties of an entity that *could have been different* **contingent**, and all properties that *cannot be different* **essential**. The **state** of an entity is just its set of contingent properties. The first distinction between values and objects is that

objects have a state and values have not.

Thus, the color, weight etc. of a car are all part of the state of a car, but being an odd prime number is not part of the state the natural number 3. The natural number 3 has no state. However, cars and numbers all have properties that may be revealed by possible observations.

Observing objects and values. The difference between objects and values is connected to the way we do observations on these entities. Properties of an object are observed by doing an *experiment*. An experiment is an event, which takes place in time, and which transfers information from the observed object to the observer. So an experiment is an observation process in which the observed object participates and whose outcome, for this particular object, could have been different than it is.

Properties of a value are observed in a *computational process*, in which information is uncovered about the observed value. The computational process consists of a manipulation of symbols, among which is a symbol that stands for the investigated value. We call this symbol a *notation* for the value. The notation is itself a physical object, with contingent properties (e.g. color for a symbol on paper, or perhaps voltage for a symbol in a computer). However, the notation *represents* a value, which has no contingent properties but has only essential properties. The computational process by which a property of a value is observed differs from an experiment, because

1. it does not involve the value to be observed but a notation for the value and
2. for the particular value that is observed, its outcome could not have been different than it is, at least if the computation is correct.

Identity of objects. An object may be in different states, which presupposes that there is *something* which is in different states. We call this something the **identifier** of the object. In general, an object identifier (oid) is an entity that is

- globally unique and
- persists through change of state.

So each car object has an identifier that is unique among all *possible* objects, not only all existing cars, but all possible cars, and even among all possible objects of any kind. The identifier of a car does not change when its color, weight, number of doors, etc. change.

Values do not have identifiers, because they do not change state. There is no identity that is presupposed to remain preserved under changes of state, for values have no state.

A consequence of the global uniqueness of identifiers is that different objects may be indistinguishable but different. Two objects may allow the same observations because they are in the same state, but if the identifiers are different, then the objects are different. This is so even if the identifiers cannot be observed. This is impossible for values v_1 and v_2: if two values cannot be distinguished by observations, then they are identical and we have $v_1 = v_2$.

Existence. Our statement that objects are observed in an experiment presupposes the fact that *objects exist in time* rather than outside time. An object may exist or it may not exist; its existence is contingent. This is not so for values: it does not make sense to say that a value exists or does not exist. We consider this to be a second essential feature of objects, and we formulate it without referring to time:

objects must exist in order to have a state.

This implies that an object is born before it has a state, and that it may die and after that cannot be said to be in a state.

This allows us to make the contingency of object state more precise. Saying that an existing object has a contingent property is saying that it *could have existed with other properties.* So a blue car could have existed as a red car. This does not imply that there ever is a state change from blue to red car, and it does not even imply that such a change is possible. It may be impossible because the laws of the country forbid it. Thus, to have an unchangeable property does not mean to have an essential property. An unchangeable property could still have been different. Rather, object has a property essentially if it could not have existed without having that property.

For example, if c is a car object, then an essential property of c is that it is a car. It could not exist without being a car. In a country that forbids painting cars, if c is a blue car object, then it will forever be a blue car object. Still, blueness is a contingent property, because c could have existed as a red car. There is a difference between essential properties and properties that, once an object has them, are fixed.

1.3 Objects and values in databases and programming languages

The idea of an identifier that keeps indistinguishable objects apart, and that preserves identity through change, is well-known in programming languages and databases. A variable in a program is identified by a name, which is its identifier. This keeps constant through changes of value and can keep equal values apart. In a database, a globally unique database key identifies records by preserving identity through changes of attribute values and it keeps records with equal attribute values apart.

The difference between the program and the database is that in a program, the programmer has to write down the identifier. This means that there are only a small number of them, and that is too limited for database applications. We can get around this by storing the identifier as value of a pointer variable and letting the program generate a new one when

necessary (e.g. using the $new(p)$ construct of Moldula). We need constructions like this, because we need an unbounded number of oid's in many applications.

However, identifiers manipulated as pointer values in a program cease to exist once the program execution is finished. In database applications, we need object lifetimes that are longer than program execution lifetimes. This can be realized by globally unique database keys, which are usually generated by the DBS itself so that we can have indefinitely many of them, and which survive the lifetime of application program executions.

Note that in a database context, observing an object is querying that object about its state or identity. Observing a value, on the other hand, is finding out what an operation does to it and is something we commonly do in a programming language. These are very different things, that are implemented in different types of systems. Observing an object is something done with the help of a DBS, but observing a value is something done with the help of a programming language. These implement different computational paradigms, that are hard to integrate. See for example [1] for a short listing of the issues involved. Embedding a DB query language in a programming language is always awkward and tends to lead to an amalgam that combines the disadvantages of both computational paradigms.

In CMSL, we use a single paradigm to do computations as is common in programming languages and to do database queries as is common in query languages. CMSL is thus an effort to eliminate the mismatch between current programming languages and database languages, sometimes called "impedance mismatch".

CMSL also integrates DB specification language and a DB query and update language. Thus, the traditional separation of a Data Definition Language (DDL) and a Data Manipulation Language (DML) is eliminated.

A DBS simulates the model by manipulating a CM specification. This symbol-manipulation process is a process in which physical *objects* are manipulated to simulate the properties of abstract values and abstract objects in a CM. The details of the simulation relation between the DBS and the CM it manipulates do not concern us here. However, we do want to note that one of the advantages mentioned for object-oriented DB's, the elimination of redundancy by object sharing, is an implementation matter that concerns the DBS only. If an attribute value of an object is another object (e.g., the father of a person object is also a person object), then the DBS stores the identifier of the father, not the object itself. This makes any updates to the father object immediately available to all objects that have the father identifier as attribute value. Storing the object state with its identifier as attribute values would create a redundancy, which in turn creates a need for multiple updates of the state of the same object, stored in different places.

1.4 Objects and values in the CM

Oid's can be implemented as global database keys. However, global database keys are an implementation concept and belong to the world of DBS's and not to CM's. What we want to do is to extract the logic of objects and their identifiers from the account given above and make this explicit in the CM specification language, alongside with the logic of values. We then have a CM that consists of objects and values, with a specification language and a logic to reason about this, as illustrated in figure 1.3. At each moment, the objects in the object system are in a state, and this is represented by abstract objects in a certain state in the CM. $(c, (color : red, weight : 3000, \ldots))$ represents the car with identifier c in a state in which it is red, has a weight of 3000 pounds, etc.

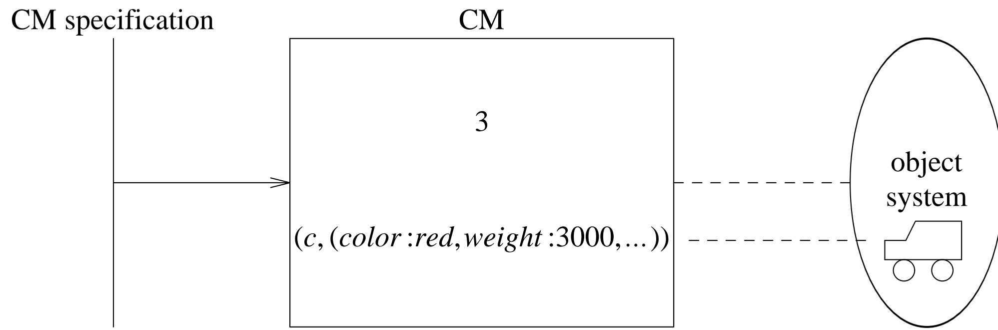

Figure 1.3: Objects and values in a CM.

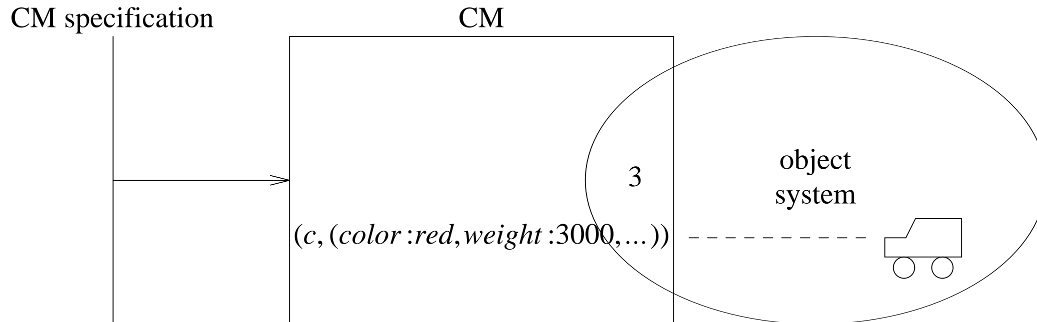

Figure 1.4: Intersection between the object system and its CM.

Note that the value 3 does not represent anything in the object system. Rather, it is just an abstract entity that is part of the conceptual model shared by people who live in the object system and/or who are experts of the object system. Usually, the part of the CM that consists of values is also part of the object system. It does not represent anything in the object system, it is a formalized part of the object system. Figure 1.4 illustrates this. In addition to the values, the abstract representation of physical objects may also be part of the intersection of the CM and the object system. To the extent that people living in the object system know the abstract representation in the CM of the physical objects in the object system, these abstract representations are part of the object system. But whatever may be the case, physical objects like cars are not part of the CM of the object system.

Chapter 2

The value specification language VSL

Values are specified in CMSL as abstract data types (ADT's). The part of CMSL in which to specify ADT's is called VSL, for Value Specification Language.

2.1 Specification

2.1.1 Introduction

VSL is based on ASF [4] and OBJ [9]. Appendix A contains the syntax of VSL and appendix B contains a number of ADT specifications in VSL. They were tested and improved by Paul Spruit and Cees Duivenvoorde, using the syntax-directed editor they built for CMSL [19].

The specifications have a structure that can be described intuitively as follows. Each specification may introduce a number of *sorts*, which are sets of values, and in may introduce a number of *functions* on those sorts. The meaning of the functions is determined by a set of *equations* for those functions. Two terms are equal if they can be proved equal using equational logic (roughly, first-order logic with equality), otherwise they are different. So in the specification `Naturals`, the terms `inc(0) + inc(inc(0))` are equal to the term `inc(inc(inc(0)))`. In usual notation, in `Naturals`, $1 + 2 = 3$.

The computational paradigm for equational specifications is *term rewriting*. Using this paradigm, a term like `inc(0) + inc(inc(0))` can be rewritten to `inc(inc(inc(0)))`.

The semantics of equational value specifications is called *initial*, which means, roughly, that

1. all elements of all sorts are named by closed terms of the sorts, and
2. closed terms name equal sort elements iff they can be proved equal from the equations in the specification.

See [5, 6, 10] for initial algebra semantics of ADT specifications.

The specification of the natural numbers and integers in appendix B is based on a specification given by Smolka et al. [16]. The number sorts form a poset illustrated in figure 2.1.

In general, VSL specifications can be parametrized and can contain hidden functions. See the specifications of sets of numbers as examples of parametrized specifications.

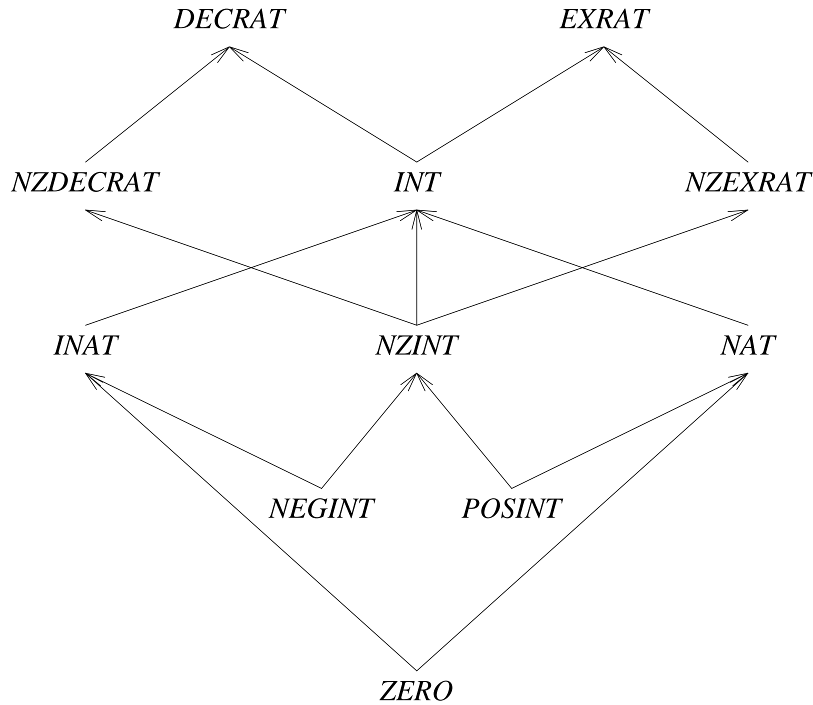

Figure 2.1: A number hierarchy.

2.1.2 Research problems

- As shown in the specification called `Naturals`, infix operators have names that start and end with a dot, except for a few common ones such as + and *. This has to do with the implementation of the editor and should be removed.
- A bug in the current syntax is that only parametrized specifications can be renamed on import.
- Currently, only the names of the imported specifications are given, and all names from the imported specification are made available. The ability to specify a list of operation names for each imported specification would be useful. Only the listed names would then be made available on import. This mechanism would be added to the hiding mechanism. Anyway, the import list of specification names currently gives no clue as to which function is declared where.
- Hidden functions should be added in the next version of CMSL.
- Renaming is not yet a signature morphism, because it is not defined what happens with the partial ordering when an element of the poset of sort names is renamed. This must be corrected.
- The ability to import externally defined functions would be useful, e.g. functions defined in C [21].
- The term rewriting paradigm for computation is inefficient for the number specifications, and moreover, we would like to have available all the constants of these data types that a particular implementation supplies. This would mean that all constants

0, 1, 2, etc. should be declared in `Naturals`, and that equations like `succ(0) = 1`, `succ(1) = 2`, etc. should be added for all these constants. This would give very large specifications whose implementation is very inefficient. An implementation of CMSL should therefore provide built-in number specifications, providing all desired constants and using an efficient implementation technique. Probably, `Dates`, `Times`, and `Money` should also be built-in.

- CMSL should be extended to include logic programming, so that narrowing can be used as operational semantics.

2.2 Interaction

2.2.1 Introduction

Once we have specified a CM consisting of values, we should be able to interact with it. A DBS should simulate this interaction. There are two ways to interact with a CM consisting of values only,

1. find the normal form of a term in an ADT specification, and

2. solve a system of equations in an ADT specification.

Any DBS should therefore provide this functionality.

The following gives an example, in a rudimentary syntax, of how such interactions could look like. A line that starts with a `v` initiates a set of declarations. A line that starts with a `q` initiates a *query*, i.e. a set of equations to be solved. A line that starts with an `e` starts an *evaluation*, i.e. a set of terms that must be normalized, where the terms range over the solution set of the equations last solved. A workbench for CMSL should provide these functions through a more sophisticated user interface.

```
begin interaction Example1
import
        Naturals, Integers
v
        x, y : NAT
q
        x + y = 5
e
        x, y, 2 * x
q
        x > 2 = true
e
        x, y
v
        x : NAT
e
        x, y
suspend Example1
```

We show the effect of this interaction by commenting on each line.

```
begin interaction Example1
import
        Naturals, Integers
```

This makes specifications available.

```
v
        x, y : NAT
```

Variables x and y are declared now.

```
q
        x + y = 5
```

The solution set of this equation is computed.

```
e
        x, y, 2 * x
```

This gives the output

```
        x  y  2*x
        ---------
        0  5   0
        1  4   2
        2  3   4
        3  2   6
        4  1   8
        5  0   10
```

```
q
        x > 2 = true
```

now the query consists of two equations. The interaction works cumulatively.

```
e
        x, y
```

The solution set is now

```
        x  y
        ----
        3  2
        4  1
        5  0
```

2.2.2 Research problems

- Details of the functionality and user interface of a workbench for VSL are currently under study [11]. Implementation of this workbench requires a study of narrowing strategies for order-sorted specifications.

Chapter 3

The object specification language OSL

Objects are specified in CMSL as abstract object types (AOT's). The part of CMSL in which objects can be specified is called OSL, for Object Specification Language. OSL includes VSL. Thus, if one wants, all ADT's could be specified in OSL. More commonly, one will specify ADT's in VSL and import these specifications in object specifications.

OSL can be used to specify attributes, events, and life cycles of objects. There are two kinds of specifications in OSL, object specifications and process specifications. Admissible import relations between these specifications are shown in figure 3.1.

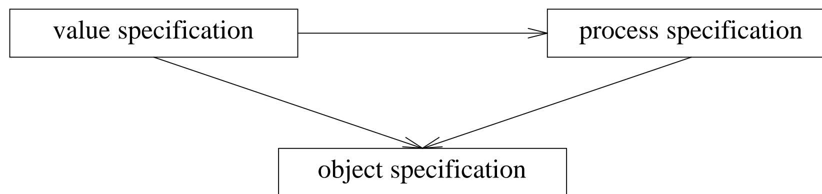

Figure 3.1: Import relations between OSL specification.

3.1 Attributes

3.1.1 Specification

The sorts defined in object specifications are interpreted just as the sorts are interpreted in values specifications. If we wish, we can define the sorts needed in an object separately in avalue specification. An example is the following value specification of the sort of address identifiers.

```
value spec AddressIdentifiers
    import
        Booleans
    sorts
        ADDRESS
    functions
        a0 : ADDRESS
```

```
        next : ADDRESS -> ADDRESS
        eq : ADDRESS x ADDRESS -> BOOL
    variables
        a, a1, a2 : ADDRESS
    equations
[1]     a eq a = true
[2]     a0 eq next(a) = false
[3]     next(a) eq a0 = false
[4]     next(a1) eq next(a2) = a1 eq a2
end spec AddressIdentifiers
```

There are infinitely many closed terms of the form `next(... next(a0) ...)` of sort `ADDRESS`. We say that `ADDRESS` is generated by `a0` and `next`. Each term of sort `ADDRESS` is a different address identifier, for there are no equations that make two different terms equal. The function `eq` is a test for equality and difference of object identifiers that yields the same result as the = sign. It is convenient to have such a function, and we usually need an `eq` function for every identifier sort.

A simple object specification, using `AddressIdentifiers` is

```
object spec Addresses
    import
        Strings, Naturals, AddressIdentifiers
    attributes
        city : ADDRESS -> STRING
        zip  : ADDRESS -> NAT
        street : ADDRESS -> STRING
        nr : ADDRESS -> NAT
end spec Addresses
```

This is a simple specification consisting of a signature, i.e. a declaration of functions, only. The intended model consists of a set of possible worlds, which all contain the ADT's specified in the value specifications. (Formally, this is called the *reduct requirement*, details of which are given in [26].) Informally, it means merely that `Addresses` contains the ADT's specified by `Strings`, and a term like `city(a)` has an element of `STRING` as value. There are no constraints on the attribute values of `ADDRESS` in a possible world other than that they must lie in their result sorts.

Note that the worlds only differ in their interpretations of the attributes. The sort `ADDRESS` and the other sorts declared in the value specifications have exactly the same extension in all possible worlds, which is the extension specified in the ADT specifications, under the initial semantics.

The attributes are total functions, and the domain of an attribute is called an **identifier sort.** This is because we may think of one world as a set of tuples of the form

```
(a, (city: c, zip: z, street: s, nr: n)),
```

where `a`, `z`, `s`, and `n`, are all elements of the appropriate sorts. The set of all these tuples for a particular `a` is called the **object** identified by `a`. The tuple shown above is this object in a certain state.

This means that we chose to view all these tuples as the *same* (identical) object, in different states. This is the view taken by certain applications in public administration. For

example, street numbers may be reassigned, zip codes may be changed, etc., in which case no one changes address, but the zip code of everyone may be changed.

We also have the freedom to distinguish addresses even if they agree on all attributes. Two persons may, for example, have legally different address and still live in the same house (for example, a house with one entrance but in a building which accommodates two different floors, which must be counted as different living addresses).

If we choose to ignore the possibility to reassign zip codes or of indistinguishable but different addresses, then we can model an address as a tuple in an ADT specification in VSL:

```
value spec AddressesAsValues
    import
        Strings, Naturals
    sorts
        ADDRESS
    functions
        (_, _, _, _) : STRING x NAT x STRING x NAT -> ADDRESS
        city : ADDRESS -> STRING
        zip  : ADDRESS -> NAT
        street : ADDRESS -> STRING
        nr : ADDRESS -> NAT
    variables
        a : ADDRESS
        c, s : STRING
        z, nr : NAT
    equations
[0]     (city(a), zip(a), street(a), nr(a)) = a
[1]     city((c, z, s, n)) = c
[2]     zip((c, z, s, n)) = z
[3]     street((c, z, s, n)) = s
[4]     nr((c, z, s, n)) = n
end spec AddressesAsValues
```

`(_, _, _, _)` is a mixfix operator that forms address tuples. The equations define the (former) attributes to be projection functions. This is a value specification, and therefore has an initial semantics. There is no way to represent the preservation in address identity when, say, zip codes are reassigned. Furthermore, difference in addresses must be represented as difference in one of the four address components.

We next define person objects. This specification is written in OSL, and imports an object specification (`Addresses`) and a value specification (`Booleans`). It also adds an ADT, that of `PERSON` identifiers, and uses VSL syntax to do that. This is allowed, because OSL contains VSL. The VSL part of this specification is the imported specification `Booleans`, and the sections `sorts`, `functions`, and `equations`.

```
object spec Persons
    import
        Addresses, Booleans
    sorts
        PERSON
    functions
```

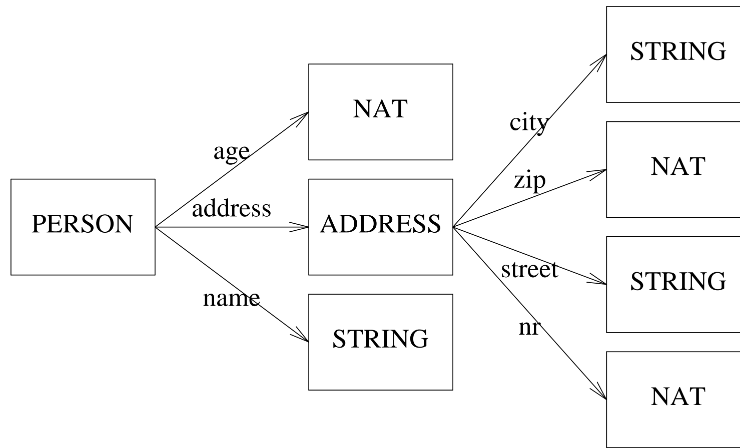

Figure 3.2: Diagram of an attribute signature.

```
        p0 : PERSON
        next : PERSON -> PERSON
        eq : PERSON x PERSON -> BOOSL
    attributes
        age : PERSON -> NAT
        address : PERSON -> ADDRESS
        name : PERSON -> STRING
    variables
        p, p1, p2 : PERSON
    equations
[1]     p eq p = true
[2]     p0 eq next(p) = false
[3]     next(p) eq p0 = false
[4]     next(p1) eq next(p2) = p1 eq p2
    constraints
[5]     age(p) < 150 = true
end spec Persons
```

This is an alternative for specifying the identifier sort `PERSON` elsewhere, in a value specification.

The `age` attribute is constrained, so that

```
(p, (age: 12, address: a, name: ``Piet''))
```

is a person state in some possible worlds, but

```
(p, (age: 160, address: a, name: ``Piet''))
```

is not.

Note that we have an identifier-valued attribute of persons, `address`. Note also that `Addresses` makes the strings and natural numbers available, so we do not have to re-import these specifications.

The signature of any specification can be shown graphically, as in figure 3.2. If we show only the attributes in a diagram, we have something resembling an EER diagram.

Existence

All attributes are total functions. Existence is represented by the simple expedient of specifying a Boolean function

```
exists : ADDRESS -> BOOL
exists : PERSON -> BOOL
```

for each identifier sort. We say that an identifier `x` *exists* iff `exists(x) = true`. See Gamut [7] for more on existence in a possible worlds semantics.

We assume `exists` implicitly declared for all identifier sorts. The value of attributes is only considered relevant on existing identifiers. So we really consider equivalence classes of worlds, where two worlds are equivalent when attributes agree on existing identifiers.

Using `exists`, we can formulate existence constraints, such as

```
when exists(p) = true
then exists(address(p)) = true
```

In general, one would want to require that identifier-valued attributes of existing identifiers have existing values. Thus, the constraint

```
when exists(p) = true
then exists(zip(p)) = true
```

is meaningless (even ill-typed), because it makes no sense of a natural number like `zip(p)` to require that it exists. Only objects can exist. Therefore, only objects can come into existence or pass away.

3.1.2 Research problems

Research problems in attribute specification are:

- Unlike value specifications, object specifications cannot be parametrized yet. This should probably be added to the language.
- The existence Boolean function should be an existence predicate. In general, we should declare attributes that are Boolean functions as predicates that are local to objects. So we should integrate logic and equational programming as well.
- Conceptual modeling methods such as EER methods and NIAM, have static constraints such as cardinality constraints, subset and exclusion constraints etc. that can be specified equationally. The diagram technique for attribute signatures should be extended with conventions to represent these constraints and standard sets of equations should be given that formalize these constraints.
- Giving syntactic criteria for local and global attribute constraints.
- Recognizing which attribute constraints are derivation rules. For example, the grandfather rule can be used as derivation rule. How can we see from a set of equational axioms that they are a definition of one function in terms of a set of other functions?

3.1.3 Implementation

A possible world can be represented by a finite set of equations that give the value of attributes on existing identifiers. For example, the world

```
(p1, (age: 12, address: a1, name: ''blah'')),
(p2, (age: 13, address: a1, name: ''John'')),
(a1, (city:  ''Amsterdam'', zip:  12345, street:
''Nieuwe Kerkstraat'', nr:  432))
```

can be represented by the set of equations

```
age(p1) = 12, address(p1) = a1, name(p1) = ''blah'',
age(p2) = 13, address(p2) = a1, name(p2) = ''John'',
city(a1) = ''Amsterdam'', zip(a1) = 12345,
street(a1) = ''Nieuwe Kerkstraat'', nr(a1) = 432.
```

Only attribute values for existing identifiers are stored. Although the example is trivial, in general the attribute constraints and ADT equations may allow a reduction of the number of equations. For example, extend the `Persons` specification with:

```
object spec PersonsWithFathers
    import
        Persons
    attributes
        father : PERSON -> PERSON
        grandfather : PERSON -> PERSON
    variables
        p : PERSON
    constraints
[1]     grandfather(p) = father(father(p))
end spec PersonsWithFathers
```

Then we need not store the equations `grandfather(p1) = p2`. Referring to figure 3.3, at any moment we have a set of elementary equations that are the *database state*, and which, together with the DB schema, have a possible world w as intended model. The DBS represents the UoD for the user, because each DBS state σ represents a CM state w, and each CM state w is an abstraction of a UoD state. The dashed triangle in figure 3.3 is therefore commutative.

w is the intended model of the DBS state σ, so the arrow from σ to w is the arrow of the interpretation function. In particular, w is the *initial algebra* of the specification consisting of the CM specification and σ (remember that σ is a set of equations). Thus, term rewriting within the specification formed by the CM specification plus σ may be used as operational semantics.

In practice, σ may be stored as a set of tuples in a relational database rather than as a set of elementary equations in a file. This is an implementation matter that does not affect picture 3.3.

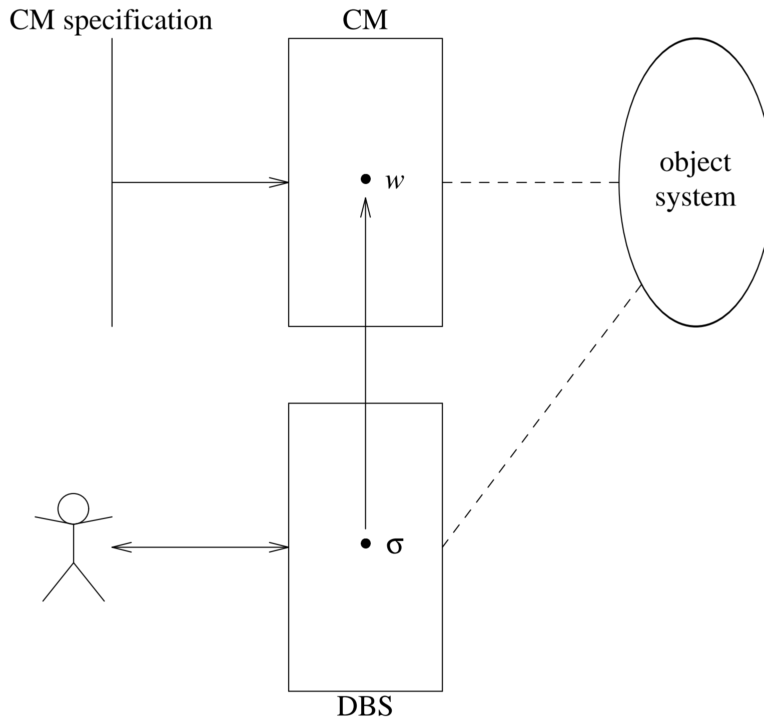

Figure 3.3: A database state.

3.1.4 Interaction

An interaction with objects is always an interaction with existing objects. This means that the variables in queries ranging over identifiers, range over existing identifiers only. It also means that an object interaction always takes place with respect to a current world. This world is pointed at by the current DB state, which is therefore made available to the interaction, together with the DB schema. An example object interaction, in the same rudimentary syntax as before, is:

```
begin interaction Example2
import
        PersonsWithFathers, DBstate
v
        p : PERSON
q
        age(p) > 15 = true
        age(p) < 25 = true
        name(city(p)) = ‘‘Amsterdam’’
e
        age(p), name(p), zip(address(p))
suspend interaction Example2
```

`PersonsWithFathers` is the DB schema in this example. `DBstate` is a set of equations that should be consistent with the DB schema and that, together with the DB schema, has a world as initial algebra. `p` ranges over existing person identifiers only.

3.1.5 Research problems

There are many research problems concerning this interaction language:

- The DB state can change during an interaction, either because during the interaction itself updates are done (see section 3.2), or because there are other interactions that run in parallel and which perform updates. It is therefore misleading to import `DBstate` explicitly, because that suggests a fixed set of equations. Another formal mechanism to select the state to be queried should be used, but what this should be, is not clear.
- The result of a query should be manipulable as a set. For example, after selecting a set of persons in `Example2`, we may want to ask for their average age.
- Higher-order functions may be needed (that accept functions as arguments and return functions as values). For example, we may want to map `age` over a set of person identifiers, as in `map(age)({ p1, ..., pn})`.
- For each of these possible extensions, the expressive power, computational complexity, and the ease of expression should be compared. An implementation should be built as well.
- Historical queries should be investigated in this framework.

3.2 Events

3.2.1 Event theory specification

Attributes are updated by events. Like attributes, events are local to an object. However, objects communicate through **messages**, which are events that cannot occur except synchronized with events executed by other objects. To specify this, we introduce a special sort `EVENT` and declare a communication operator on this sort.

```
process spec EventAxioms
    sorts
        FAIL < EVENT
    functions
        fail : FAIL
        _ & _ : EVENT x EVENT -> EVENT        [AC]
    variables
        e : EVENT
    equations
[1]     fail & e = fail
end spec EventAxioms
```

A `process spec` has the same syntax as a `value spec`. The difference is semantic, and is that a process specification does not need to have an initial semantics. below, we give an example of a process specification with a *process graph* semantics, in which there are infinite elements, that are not named by a closed term, as would be required in an initial semantics. However, any value specifications that should be imported in a process specification must still have an initial semantics.

The constant `fail` denotes the failure event that leads nowhere and that represents eternal stagnation. It is also called *deadlock* or *unsuccessful termination*. `&` is the *synchronization function*. `e1 & e2` denotes the event consisting of a synchronous occurrence of events `e1` and `e2`. `&` is associative and commutative (`[AC]` is an abbreviation for the equations defining `&` to be so). A synchronous occurrence with `fail` always fails. If it would not, then this would imply that `fail` itself led to a next state and that is a contradiction.

The structure of a CM is now as follows.

1. There is a collection of ADT's, each of which consist of sets of values.
2. There is a collection of possible worlds, which all share the ADT's.
3. There is an algebra of events, containing in particular a failure event and a synchronization operator.

The events will be interpreted as functions on possible worlds in the next section.

3.2.2 Event specification

A simple object specification with events is

```
object spec PersonEvents
    import
        PersonAttributes, EventAxioms
    events
        inc-age : PERSON -> EVENT
        change_address : PERSON x ADDRESS -> EVENT
        change_name : PERSON x STRING -> EVENT
    variables
        p : PERSON
        a : ADDRESS
        s : STRING
    constraints
[1]     when age(p) = n then [inc_age(p)] age(p) = n+1
[2]     [change_address(p, a)] address(p) = a
[3]     [change_name(p, s)] name(p) = s
end spec PersonEvents
```

The `events` section populates the sort `EVENT` with events. Terms of sort `EVENT` can only occur at the position e in modal formulas of the form $[e]\psi$. In general, the meaning of $\phi \rightarrow [e]\psi$ is "if ϕ is true in the current world, then in all the worlds to which performing e leads us, ψ is true". Note that this is *partial correctness*, for there may be no worlds to which `e` leads us. We won't bother about this here, and we will assume that all events except `fail` always terminate.

`[1]` says that the effect of `inc_age(p)` in a world where `p` is less than 149 years of age, is to increase `age(p)` by 1. `[2]` and `[3]` define the effect of `change_name` and `change_address` as expected.

It can be shown that constraint `[1]`, together with the static constraint on age (`[5]` in `Persons`), is equivalent to the set of constraints

```
[1']     when age(p) = n and n < 149 = true
         then [inc_age(p)] age(p) = n+1
[1'']     when age(p) = n and n < 149 = false
         then [inc_age(p)] false
```

[1'] says that the effect of inc_age(p) in a world where p is less than 149 years of age, is to increase age(p) by 1. [1''] says that an attempt to perform inc_age in any other world would fail, i.e. leads to no successor world.

The constraints in PersonEvents are not sufficient to be able to interpret events as functions on possible worlds, because the constraints only say what changes from the current world to the next. It is assumed that the rest of the world does *not* change and in order to interpret the effect of an event on a possible world, we must make this assumption explicit. This assumption is called the *frame assumption* and says that

> if we cannot prove a formula of the form
>
> $$a(x) = t \rightarrow [e(x)]a(x) = t',$$
>
> with t and t' terms from an ADT specification, then we assume a formula
>
> $$a(x) = t \rightarrow [e(x)]a(x) = t.$$

In other words, if we cannot prove what the new value of an attribute is after performing an event, then we assume it is unchanged.

The logic to be used for reasoning about objects with events is equational dynamic logic [13, 26].

Communication

To illustrate communication, take money transfer between bank accounts.

```
object spec Accounts
    import
        Integers, EventAxioms
    sorts
        ACCOUNT
    functions
        a0 : ACCOUNT
        next : ACCOUNT -> ACCOUNT
    attributes
        balance : ACCOUNT -> INT
    events
        send : ACCOUNT x ACCOUNT x INT -> EVENT message
        receive : ACCOUNT x ACCOUNT x INT -> EVENT message
        transfer : ACCOUNT x ACCOUNT x INT -> EVENT
    variables
        a, a1, a2 : ACCOUNT
        n, m : INT
    equations
[0]     send(a1, a2, n) & receive(a2, a1, n) = transfer(a1, a2, n)
```

```
    constraints
[1]     when balance(a1) = n
        then [send(a1, a2, m)] balance(a1) = n-m
[2]     when balance(a2) = n
        then [receive(a2, a1, m)] balance(aa2) = n+m
end spec Accounts
```

`ACCOUNT` is generated by `a0` and `next`.

An event that is flagged a **`message`** cannot be executed. This means that it is interpreted as the `fail` event. The role of messages is to define the effect of *communication* events. In equation `[0]`, `transfer` is defined as a communication event equal to the synchronous execution of a `send` and a `receive` event. This means that the effect of `transfer(a1, a2, n)` equals the joint effect of `send(a1, a2, n)` and `receive(a2, a1, n)`. Since `transfer` is not itself a message, it will not be interpreted as `fail`.

Any equation of event terms in which the operator for synchronous execution `&` occurs is called a *communication equation*, and an event equal to a term containing `&` is called a *communication event*. A communication equation is required to have a simple form, in which a communication event is defined to be equal to a term containing only messages. There must not be an equation defining one of these messages to be a communication event.

All noncommunication events have a *subject*, which is their first argument. This is the object in whose life the event occurs, i.e. who suffers or performes the event. A communication event has more than one subject, viz. the set of subjects of its messages. A communication event in which each message has a different subject is called *fully global* or simply *global*. A communication event that is not local and not fully global is called *partly global*.

Any set of events involving different subjects that can occur at all (i.e. that does not contain messages as elements) is always allowed to occur synchronously. To interpret events $e_1, \ldots, e_n$ performed or suffered by a set of objects, we consider an event term $e_1 \& \cdots \& e_n$. Two things are done with this term, in the following order:

1. The communication equations are applied to eliminate messages as far as possible.
2. If there are remaining messages, these are interpreted as `fail`. By equation `[1]` of `EventAxioms`, this means that execution of the set as a whole fails.
3. The result of this is a set $e'_1, \ldots, e'_k$ of events, some of which are noncommunication events with a single subject, and others of which are communication events with one or more subjects. If the sets of subjects of the events are all pairwise disjoint, then the event set can be executed, otherwise execution fails.

Communication is thus formalized by the message mechanism. Mere synchronous execution of two events by the operator `&` is not a transfer of information. But if one or more participating events is a message, then this event could not occur except burried inside a communication event. For example, a synchronous execution of two `inc_age` events is not a transfer of information between the two persons who suffer these events. But the synchronous execution of `send(a1, a2, n)` and `receive(a2, a1, n)` *is* a transfer of information, because these events could not occur in any other way. Thus, a synchronous execution of `send(a1, a2, n)` with `receive a3, a4, m)` would fail, unless `a1` = `a3`, `a2` = `a4`, and `n` = `m`.

Existence

There is a special object called the *existence monitor*, with identifier `db`, that performs object creation and deletion. The birth and death of an object are events that communicate with corresponding events in the life of `db`. The initial state of the created object is an argument to the creation event. An example is

```
create(db, p15, 12, ``Piet'', a20)
```

3.2.3 Research problems

- The set of events occurring in one state transition of a CM is called a *synchronicity set*. A synchronicity set may be ambiguous in that it may be parsed in several ways as set of communications. This may be an implementation matter, but it should be looked into at the level of formal semantics.

- In reality, if one object deadlocks, then other objects may continue. So `&` is not an axiomatization of synchronous execution by *different* objects, although it is a good axiomatization of synchronous execution by the same object. Interobject parallelism differs fundamentally from intraobject parallelism. This point will return when we discuss parallelism between processes.

- Events occur at specific points in time, and this information is important for databases. Real time should therefore be added to CMSL.

- Objects perform or suffer events, but the *initiative* of events is not yet modeled. Research in this is currently being done [15, 28]. This may also make it possible to model flow of information in a communication.

- `&` is really a second-order operator that accepts functions (events) as arguments and delivers a function (event). It would be interesting to formalize this in second-order equational logic and see if we can come up with a simpler communication mechanism this way.

The formalization of existence is not satisfying and should be improved at least in the following respects:

- It is always `db` that performs `create`, so it ought not to be necessary to state this explicitly. Yet, this is required if `create` is declared as a function of the right arity.

- On the other hand, `p15` is a constant that should be generated by the system. The user should not type it in, but it should be given to the user. This is not visible in the current formalization. This differs from the matter of initiative of an event, it concerns instead the initiative of providing the value of a parameter. An object creation mechanism such as proposed in [22] on the basis of the process creation operator [3] would solve this, but the current proposal seems too complicated to adopt for CMSL.

- The state vector should be labeled by the attribute names. This is probably a minor problem.

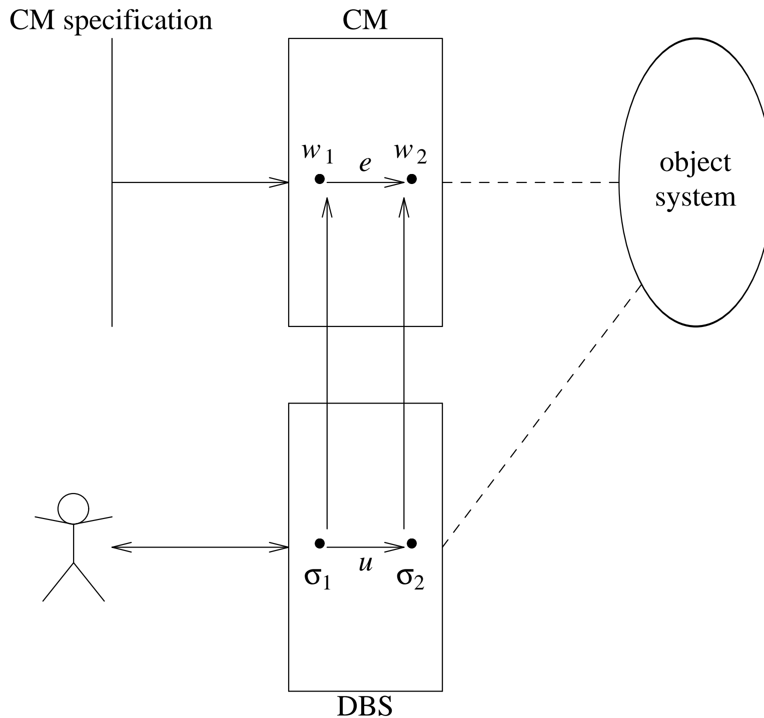

Figure 3.4: Database updates.

3.2.4 Implementation

In a DBS, an event can be simulated by a function on database states, that adds or deletes equations when moving from one state to the next. The requirement on the function on DB states is that the diagram in figure 3.4, formed by event e and update u and the two interpretation arrows on DB states, is commutative.

Interaction

A DB interaction can now also contain updates. The following gives an example of how this could look like.

```
resume interaction Example2
u
    change_name(p1, "Piet")
v
    p : PERSON
q
    zip(address(p)) = 123
u
    inc_age(p)
suspend interaction Example2
```

This resumes the example begun in section 3.1.4. `p1` is a closed term, and `change_name(p1, "Piet")` changes the current `DBstate`. Next, a query determines a set of values for `p`, and an update is performed on all objects thus identified. Note that these are existing objects.

The update must be interpreted as a parallel execution of all individual updates involved. The query and update are really a formula of the form $\&x : PERSON(\phi(x) \rightarrow e(x))$, where & acts as a binding operator just as $\forall$. The precise semantics of this is subject of current research.

3.2.5 Research problems

- Update semantics is an active research problem. Examples of open problems here are the determination of derived updates, updates with negative information, updates with incomplete information, etc.
- Another research problem is the optimization of integrity maintenance in logic databases. It is not yet known how much of this can be transferred to the equational context.
- Giving syntactic criteria for locality of event constraints.
- Should hidden events be allowed?
- We need broadcasts, in which a message is sent to a set of objects specified by a predicate.
- Querying a DB state in the context of an event specification is more complex than querying it in the context of an attribute specification, for modal formulas may be relevant for computing the solution set.

3.3 Life cycles

Objects have a life in which they perform or suffer events. We follow the custom of calling this a **life cycle**, although there is usually nothing cyclic about it and the word *process* would be more appropriate. To specify life cycles, we must first extend the event theory specification to a process theory specification and then use that specification to define the life cycle of classes of objects.

3.3.1 Process theory specification

The process theory we use is ACP (Algebra of Communicating Processes) [2]. An example process theory specification is

```
process spec ProcessAlgebra
    import
        EventAxioms
    sorts
        EVENT < PROCESS
    functions
        _ + _  : PROCESS x PROCESS -> PROCESS
        _ ; _  : PROCESS x PROCESS -> PROCESS
        _ || _ : PROCESS x PROCESS -> PROCESS
        _ |L _ : PROCESS x PROCESS -> PROCESS
        _ & _  : PROCESS x PROCESS -> PROCESS
    variables
```

```
        e, e1, e2 : EVENT
        x, y, z : PROCESS
    equations
[A1]    x+y = y+x
[A2]    (x+y)+z = x+(y+z)
[A3]    x+x = x
[A4]    (x+y);z = x;z + y;z
[A5]    (x;y);z = x;(y;z)
[A6]    x + fail = x
[A7]    fail;x = fail
[CM1]   x || y = x |L y + y |L x + x & y
[CM2]   e |L x = e;x
[CM3]   (e;x) |L y = e;(x || y)
[CM4]   (x+y) |L z = x |L z + y |L z
[CM5]   (e1;x) & e2 = (e1 & e2);x
[CM6]   e1 & (e2;x) = (e1&e2);x
[CM7]   (e1;x) & (e2;x) = (e1&e2) ; (x || y)
[CM8]   (x+y) & z = (x&z) + (y&z)
[CM9]   x&(y+z) = (x&y) + (x&z)
end spec ProcessAlgebra
```

The sort of events is extended to the sort of processes, and the synchronous execution operator `&` is extended to processes. There are no events declared in this specification, but when `ProcessAlgebra` is imported in an object specification, all terms of sort `EVENT` are events. The operators defined for processes have the following intuitive meaning. `+` denotes choice, `;` denotes sequence, `||` denotes parallel composition and `|L` denotes left merge, explained below.

Given the intuitive meaning of `+` and `;`, axioms `[A1]` through `[A5]` are easy to understand. For example, `[A1]` says that a choice between `x` and `y` is a choice between `y` and `x`. Note that there is no axiom `x;(y+z) = x;y + x;z`. In the process `x;(y+z)`, after doing `x`, there is a choice between `x` and `y`, but in the process `x;y + x;z`, after doing `x`, there is no choice. `x;y + x;z` is nondeterministic, for after doing `x`, the process may be in a state where it can do `y` or it may be in a state where it can do `z`, but it is *not* in the position to choose between `y` and `z`.

`[A6]` sais that giving a choice between failure (deadlock, eternal stagnation) and something else, a process will always choose this other branch. `[A7]` sais that after deadlock, nothing happens.

The rest of the axioms defines parallel composition. `x |L y` is the process that starts with an event from `x` and then behaves as the parallel composition of the rest of `x` with `y`. This is stated in `[CM2]` and `[CM3]`.

In `[CM5]` through `[CM7]`, `x & y` is defined as the process that starts with a synchronous execution of the first events of `x` and `y`, and then performs the parallel composition of the rest of `x` with the rest of `y`.

`[CM1]` defines the parallel composition `x || y` to be an *interleaving* of `x` and `y`. This is a process that, at each moment, chooses between a next event of `x`, a next event of `y`, or a synchronous execution of a next event from `x` with a next event from `y`. The other axioms are easy to understand.

A process can be visualized as a graph, as illustrated in figure 3.5. Graph (a) shows

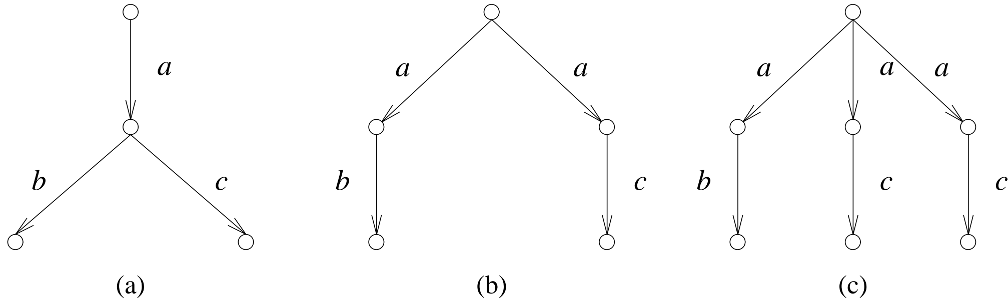

Figure 3.5: Some process graphs.

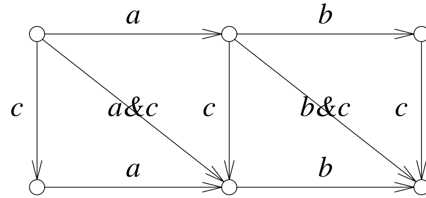

Figure 3.6: Parallel composition.

the process $a;(b+c)$ and graph (b) shows the process $a;b+a;c$. It is clear that these are different, because the choice event occurs at different places. Graph (c) shows the process $a;b+a;c+a;b$. According to axioms [A1] through [A3], this is equal to the process $a;b+a;c$, so the two graphs (b) and (c) represent the same process.

Parellel composition of processes can be represented as the Cartesian product of two graphs. Figure 3.6 shows the process $(a;b)||c$, where we assume a and b are performed by one object, and c by another. a and b can thus occur synchronously with c, as shown by the diagonals of the Cartesian product. To illustrate communication between processes, consider `send(a1, a2, n) || receive(a2, a1, n)`, declared in `Accounts`. Figure 3.7 shows the Cartesian product of the single-edge graphs for `send(a1, a2, n)` and `receive(a2, a1, n)`. Taking into consideration that `send(a1, a2, n) & receive(a2, a1, n)` equals `transfer(a1, a2, n)`, the diagonal of the graph is can be relabeled to `transfer(a1, a2, n)`. Adding the fact that `send(a1, a2, n)` and `receive(a2, a1, n)` separately are equal to `fail`, this is the only edge of the graph that remains, for [A6] allows us to drop the edges labeled `fail`.

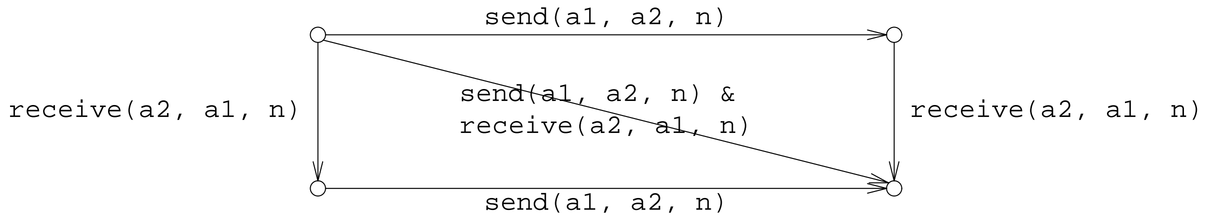

Figure 3.7: Communication.

3.3.2 Research problems

- Parallel composition should be represented in a more perspicuous way, e.g. using a mechanism like Harel's state charts [12].
- Processes should be representable at different levels of detail. For example, we should be able to further specify an edge in a process graph in another process graph. Research on this is currently being done [17, 20].
- Object-oriented models need two semantics of parallelism, *true concurrency* for parallelism between objects, *interleaving* for parallelism in a single object. There are true concurrency models for process algebra, and interleaving models [8], but as yet there is no model that mixes these two.
- Currently, we allow only events inside the box operator. An interesting avenue of research is the extension of this by allowing finite and even recursively defined (e.g. through a least fixpoint operator) process terms inside the box operator. A hard case is the possibility of parallel composition inside the box, which is not compositional. Given the modularity of inter-object parallelism, something may be done in this direction, though.
- Processes are interpreted as relations on possible worlds. They are thus modal operators, interpreted as accessibility relations on worlds. Process operators like `+`, `;` and `&` are really second-order operators that map modal operators to modal operators. This is not really formalized in the current version of CMSL but seems an interesting topic for further research.

3.3.3 Life cycle definition

An example of a life cycle definition for an object is:

```
object spec AccountLifeCycle
    import
        Accounts, ProcessAlgebra, Persons
    attributes
        owner : ACCOUNT -> PERSON
    events
        openAccount : PERSON x ACCOUNT -> EVENT  message
        closeAccount : PERSON x ACCOUNT -> EVENT  message
        accountOpened : ACCOUNT x PERSON -> EVENT message
        accountClosed : ACCOUNT x PERSON -> EVENT  message
        open : ACCOUNT x PERSON -> EVENT
        close : ACCOUNT x PERSON -> EVENT
    variables
        a : ACCOUNT
        p : PERSON
    equations
[1]     openAccount(p, a) & accountOpened(a, p) = open(a, p)
[2]     closeAccount(p, a) & accountClosed(a, p) = close(a, p)
    life cycle
[1]     ACCOUNT = openAccount ; ACCOUNT_LIFE ; closeAccount
```

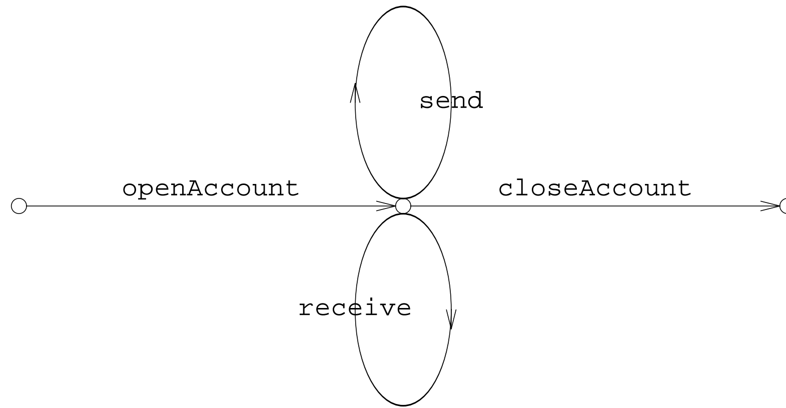

Figure 3.8: The account life cycle.

```
[2]      ACCOUNT_LIFE = (send + receive); ACCOUNT_LIFE
end spec AccountLifeCycle
```

The `life cycle` definition is a set of equations that is solved in the process model. The capitalized words in it are variables of sort `PROCESS`, the other words in it are events. There must be at least one process variable whose name is the name of an identifier sort. That variable will denote the life cycle performed or suffered by instances of the class of objects. So the `ACCOUNT` process defined in the life cycle definition above defines the life cycle of `ACCOUNT` instances.

The life cycle definition is said to *specify* the set of processes satisfying the equations, although *define* would be a better word. If the set of equations is *guarded*, there is exactly one solution. Roughly, this means that every variable x at the right-hand side must occur in a (sub)term of the form $a; x$, where a is an event. A graph for the solution of the `ACCOUNT` life cycle definition is shown in figure 3.8.

If there is no explicit life cycle definition for an identifier sort, such as for `PERSON`, then we assume that that life cycle is an unstructured iteration over all its possible events. So the person life cycle is

```
PERSON = (inc_age + change_address + change_name +
          openAccount + closeAccount) ; PERSON
```

3.3.4 Research problems

- It is not clear what the relation is between the life cycle of a class and the life cycles of its specializations. The life cycle for a car is a specialization of the life cycle of a vehicle , but for every proposed formalization of this relation, counterexamples have been found. Research on this topic continues.

- At any moment, a CM is in a state determined by the set of objects that then exist and the state of those objects. This CM state always includes the trace of events that led to it. Many applications require storing this trace, to facilitate historical queries. The logic for this kind of query is probably a version of temporal logic that is able to pose questions about the past and possible future. The structure upon which a temporal logic query operates is, precisely, a process graph defined by a life cycle definition. An extension of CMSL with temporal logic should be studied in the future.

- As said before, when discussing extensions to the event specification mechanism, real-time should be added, to allow time-stamping of event occurrences in a trace of events.
- The life cycle of an object starts with birth (`exists(p)` becomes true) and ends with death (`exists(p)` becomes false). The current formalization of this is that there is a special object, called the *existence monitor*, that creates an object before it performs or suffers its first event and possibly deletes an object after it performed or suffered its last event. This turns out to be a quite complicated affair, especially if one takes existence constraints into account, and simpler solutions should be sought.

3.3.5 Implementation

A DBS simulates a CM by simulating the life cycle of all objects in it. The life cycle definition is then a dynamic constraint on the order of event ocurrences.

Chapter 4

The conceptual model specification language (CMSL)

4.1 Specification

A CM specification in CMSL just imports the required value- and object- and process specifications. The only thing added to this is a specification of the initial state of the model. By default, the existence monitor is the only object that exists, but other objects may be specified to exist as well. The initial state is specified by a finite set of equations. An example CM specification is

```
conceptual model spec Example
    import
        Persons, AccountLifeCycle
    initialization
[1]     exists(db) = true
[2]     exists(p0) = true
[3]     exists(a0) = true
[4]     age(p0) = 30
[5]     name(p0) = ``Piet''
[6]     address(p0) = a0
[7]     city(a0) = ``Amsterdam''
[8]     zip(a0) = 1234
[9]     street(a0) = ``De Boelelaan''
[10]    nr(a0) = 1081
end spec Example
```

4.2 Implementation

The finite set of equations in the initialization is a database state, and can therefore be implemented as a database state or even a file that is loaded into a database. The CMSL implementation must then verify that the set of equations specified a possible world in the CM specified by the CMSL specification.

Appendix A

BNF grammar of CMSL

The syntax of CMSL is written in a context free syntax using the following notational conventions and abbreviations (cf. [4]).

- ¡Everything typed between angles is a nonterminal¿.
- Single-character terminal symbols like "[" are written between quotes.
- ¡nonterminal¿* denotes 0 or more occurrences of ¡nonterminal¿.
- ¡nonterminal¿ ¡sep¿* denotes 0 or more occurrences of ¡nonterminal¿ separated by ¡sep¿.
- [¡nonterminal¿] denotes 0 or 1 occurrences of ¡nonterminal¿.
- ¡nonterminal¿+ denotes 1 or more occurrences of ¡nonterminal¿.
- ¡nonterminal¿ ¡sep¿+ denotes 1 or more occurrences of ¡nonterminal¿ separated by ¡sep¿.

A.1 VSL

```
<value specification> ::= value spec <identifier>
                              [<parameters>]
                              [<imports>]
                              [<sorts>]
                              [<functions>]
                              [<variables>]
                              [<equations>]
                          end spec [<identifier>]

<parameters> ::= parameters {<parameter> ","}+

<parameter> ::= <identifier> begin
                    [<sorts>]
                    [<functions>]
                end <identifier>

<imports> ::= imports {<module-expression> ","}+
```

```
<module-expression> ::=
    <identifier> using {<identifier> for <identifier>
                [binding "[" {<bindings> ", " }+"]" ] "," }*
                [renaming "[" {<renamings> ", "}+ "]" ]

<bindings> ::= {<identifier> "->" <target>", "}+

<target> ::= <identifier> | <product>

<product> ::= {<identifier> " x "}+

<renamings> ::= {<identifier> "->" <identifier> ", "}+

<sorts> ::= sorts {<subsorts> ", "}+

<subsorts> ::= <identifier> | {<identifier> "<" }+

<functions> ::= functions {<function> [hidden]}+

<function> ::= <identifier> ":" <identifier> |
    <function or operator> ":" <product> "->" <product>

<function or operator> ::= <identifier> | <operator>

<operator> ::= <identifier> " _ " |
               " _ " <identifier> " _ "

<variables> ::= variables {<declaration>}+

<declaration> ::= {<identifier> ", ">}+ ":" <identifier>

<equations> ::= equations {<tagged equation>}+

<tagged equation> ::= <tag> <equation>

<tag> ::= "[" <identifier> "]"

<equation> ::= <unconditional equation> |
               <conditional equation>

<unconditional equation> ::= <term> "=" <term>

<conditional equation> ::=
    when {<unconditional equation> and}+
    then <unconditional equation>

<term> ::= <identifier> |
           <tuple> |
```

```
          <application>

<tuple> ::= "<" <terms> ">"

<terms> ::= {<term> ", "}+

<application> ::= <function application> |
                  <operator application>

<function application> ::= <identifier> "(" <terms> ")"

<operator application> ::= {operator application part}+

<operator application part> ::= <identifier> | <term>
```

A.2 OSL

```
<process specification> ::= process spec <identifier>
                              [<parameters>]
                              [<imports>]
                              [<sorts>]
                              [<functions>]
                              [<variables>]
                              [<equations>]
                          end spec [<identifier>]


<object specification> ::= object spec <identifier>
                             [<parameters>]
                             [<forward>]
                             [<imports>]
                             [<sorts>]
                             [<functions>]
                             [<attributes>]
                             [<events>]
                             [<variables>]
                             [<equations>]
                             [<constraints>]
                             [<life cycle>]
                             [<comment>]
                         end spec <identifier>

<forward> ::= forward {<parameter> ", "}+

<identifier sorts> ::= identifier sorts
                  <particulars> {specializing <particulars>}
```

```
<particulars> ::= <identifier> | "{" {<identifier>}+ "}"

<attributes> ::= attributes {<attribute>}+

<attribute> ::= <identifier> ":" <identifier> "->" <product>

<events> ::= events {<event>}+

<event> ::= <identifier> ":" <product> "->" <identifier> [<event type>]

<event type> ::= local message | global message | final event

<constraints> ::= {<tagged equation>}+

<life cycle> ::= life cycle
                  {<unconditional equation>}+

<comment> ::= ";" " " <text> <newline>
```

A.3 CMSL

```
<specification> ::= <value specification>*
                    <process specification>*
                    <object specification>*
                    <CM specification>*

<CM specification> ::= conceptual model spec <identifier>
                           <imports>
                           [<initializations>]
                       end spec <identifier>

<initializations> ::= initialization
                           {<tag> <unconditional equation>}+
```

Appendix B

Example ADT specifications

The following example specifications have been checked by a syntax-checking editor, CME, built by Paul Spruit and Cees Duivenvoorde [19].

```
value spec Booleans
  sorts
    BOOL
  functions
    true           : BOOL,
    false          : BOOL,
    .not. _        : BOOL        -> BOOL,
    _ .and. _      : BOOL x BOOL -> BOOL,
    _ .or. _       : BOOL x BOOL -> BOOL,
    _ .implies. _ : BOOL x BOOL -> BOOL,
    _ .equiv. _    : BOOL x BOOL -> BOOL
  variables
    x, y, z : BOOL
  equations
[A1]  true .and. x = x
[A2]  false .and. x = false
[N1]  .not. true = false
[N2]  .not. false = true
[O]   x .or. y = .not. ((.not. x) .and. (.not. y))
[I]   x .implies. y = (.not. x) .or. y
[E]   x .equiv. y = (x .implies. y) .and. (y .implies. x)
end spec Booleans

value spec Naturals
  imports
    Booleans
  sorts
    ZERO < NAT, POSINT < NAT
  functions
    0         : ZERO,
    1         : POSINT
    inc       : NAT           -> POSINT,
```

```
    dec        : POSINT       -> NAT,
    _ + _      : NAT x NAT    -> NAT,
    _ * _      : NAT x NAT    -> NAT,
    _ .div. _  : NAT x POSINT -> NAT,
    _ ^ _      : NAT x NAT    -> NAT,
    _ <= _     : NAT x NAT    -> BOOL,
    _ < _      : NAT x NAT    -> BOOL,
    _ >= _     : NAT x NAT    -> BOOL,
    _ > _      : NAT x NAT    -> BOOL,
    _ .eq. _   : NAT x NAT    -> BOOL,
    _ .neq. _  : NAT x NAT    -> BOOL
  variables
    m, n : NAT
    p, q : POSINT
  equations
[1]   1 = inc (0)
[D1]  dec (inc (n)) = n
[D2]  inc (dec (p)) = p
[A1]  n + 0 = n
[A2]  n + inc (m) = inc (n + m)
[M1]  n * 0 = 0
[M2]  n * inc (m) = (n * m) + n
[DI1] when p > m = true then m .div. p = 0
[DI2] when p .eq. m = true then m .div. p = 1
[DI3] when (p < m) = true and p + q = m
      then m .div. p = 1 + (q .div. p)
[E1]  m ^ 0 = 1
[E2]  m ^ inc (n) = m * (m ^ n)
[L1]  0 < 0 = false
[L2]  0 < p = true
[L3]  p < 0 = false
[L4]  (inc (n) < inc (m)) = (n < m)
[G]   (n > m) = (m < n)
[LE]  (n <= m) = .not. (m < n)
[GE]  (n >= m) = .not. (n > m)
[E]   (n .eq. m) = ((n <= m) .and. (m <= n))
[NE]  n .neq. m = .not. (n .eq. m)
end spec Naturals

value spec Integers
  imports
    Naturals
  sorts
    POSINT < NAT < INT,
    NEGINT < INAT < INT,
    ZERO < INAT,
    NEGINT < NZINT < INT,
    POSINT < NZINT
```

```
  functions
    dec       : INAT            -> NEGINT,
    inc       : NEGINT          -> INAT,
    dec       : INT             -> INT,
    inc       : INT             -> INT,
    _ + _     : INT x INT       -> INT,
    _ - _     : INT x INT       -> INT,
    neg       : INT             -> INT,
    neg       : NZINT           -> NZINT,
    _ * _     : INT x INT       -> INT,
    _ * _     : NZINT x NZINT   -> NZINT,
    _ * _     : POSINT x POSINT -> POSINT,
    _ <= _    : INT x INT       -> BOOL,
    _ < _     : INT x INT       -> BOOL,
    _ >= _    : INT x INT       -> BOOL,
    _ > _     : INT x INT       -> BOOL,
    _ .eq. _  : INT x INT       -> BOOL,
    _ .neq. _ : INT x INT       -> BOOL,
    gcd       : NZINT x NZINT   -> NZINT
  variables
    i, j : INT
    p, p1, p2, p3 : POSINT
    n : NEGINT
  equations
[D1]   inc (dec (i)) = i
[D2]   dec (inc (i)) = i
[A1]   i + 0 = i
[A2]   i + inc (j) = inc (i + j)
[A3]   i + dec (j) = dec (i + j)
[S1]   i - 0 = i
[S2]   i - inc (j) = dec (i - j)
[S3]   i - dec (j) = inc (i - j)
[N]    neg (i) = 0 - i
[M1]   i * 0 = 0
[M2]   i * inc (j) = (i * j) + i
[M3]   i * dec (j) = (i * j) - i
[L1]   0 < 0 = false
[L2]   0 < n = false
[L3]   0 < p = true
[L4]   inc (i) < j = i < dec (j)
[L5]   dec (i) < j = i < inc (j)
[G]    (i > j) = (j < i)
[LE]   (i <= j) = .not. (j > i)
[GE]   (i >= j) = .not. (i < j)
[E]    (i .eq. j) = ((i <= j) .and. (j <= i))
[NE]   i .neq. j = .not. (i .eq. j)
[GCD1] gcd (p, p) = p
[GCD2] gcd (n, p) = gcd (neg (n), p)
```

```
[GCD3] gcd (p, n) = gcd (p, neg (n))
[GCD4] gcd (n, n) = n
[GCD5] when p1 < p2 = true and p2 = p1 + p3
       then gcd (p1, p2) = gcd (p1, p3)
[GCD6] when p1 > p2 = true and p2 = p1 + p3
       then gcd (p1, p2) = gcd (p3, p2)
end spec Integers

value spec ExactRationals
  imports
    Integers
  sorts
    NZINT < NZEXRAT < EXRAT, INT < EXRAT
  functions
    _ + _      : EXRAT x EXRAT     -> EXRAT,
    _ - _      : EXRAT x EXRAT     -> EXRAT,
    _ * _      : EXRAT x EXRAT     -> EXRAT,
    _ / _      : EXRAT x NZEXRAT   -> EXRAT,
    _ / _      : NZEXRAT x NZEXRAT -> NZEXRAT,
    num        : EXRAT             -> INT,
    denom      : EXRAT             -> NZINT,
    _ <= _     : EXRAT x EXRAT     -> BOOL,
    _ < _      : EXRAT x EXRAT     -> BOOL,
    _ >= _     : EXRAT x EXRAT     -> BOOL,
    _ > _      : EXRAT x EXRAT     -> BOOL,
    _ .eq. _   : EXRAT x EXRAT     -> BOOL,
    _ .neq. _  : EXRAT x EXRAT     -> BOOL
  variables
    i, j : INT
    nz, nz1, nz2 : NZINT
    n1, n2 : NEGINT
    p, p1, p2 : POSINT
    r, r1, r2 : EXRAT
  equations
[F0]  num (r) / denom (r) = r
[F1]  num (i / nz) = i
[F2]  denom (i / nz) = nz
[D1]  i / 1 = i
[D2]  0 / nz = 0
[D3]  neg (p) / nz = p / neg (nz)
[D4]  when gcd (p, nz) .neq. 1 = true
      then p / nz = (1 / gcd (p, nz)) / (nz / gcd (p, nz))
[A]   (i / nz1) + (j / nz2) = ((i * nz2) + (j * nz1)) / (nz1 * nz2)
[S]   (i / nz1) - (j / nz2) = ((i * nz2) - (j * nz1)) / (nz1 * nz2)
[M]   (i / nz1) * (j / nz2) = (i * j) / (nz1 * nz2)
[L1]  (i / p1) < (j / p2) = (i * p2) < (i * p1)
[L2]  (i / p1) < (j / n2) = (i * n2) > (i * p1)
[L3]  (i / n1) < (j / p2) = (i * p2) > (i * n1)
```

```
[L4]  (i / n1) < (j / n2) = (i * n2) < (i * n1)
[G]   (r1 > r2) = (r2 < r1)
[LE]  (r1 <= r2) = .not. (r2 > r1)
[GE]  (r1 >= r2) = .not. (r1 < r2)
[E]   (r1 .eq. r2) = ((r1 <= r2) .and. (r2 <= r1))
[NE]  r1 .neq. r2 = .not. (r1 .eq. r2)
end spec ExactRationals

value spec DecimalRationals
  imports
    ExactRationals
  sorts
    NZINT < NZDECRAT < DECRAT, INT < DECRAT
  functions
    _ . _      : INT x NAT       -> DECRAT,
    intpart    : DECRAT          -> INT,
    decpart    : DECRAT          -> NAT,
    floor      : EXRAT           -> DECRAT,
    dec2ex     : DECRAT          -> EXRAT,
    _ + _      : DECRAT x DECRAT -> DECRAT,
    _ - _      : DECRAT x DECRAT -> DECRAT,
    _ * _      : DECRAT x DECRAT -> DECRAT,
    _ <= _     : DECRAT x DECRAT -> BOOL,
    _ < _      : DECRAT x DECRAT -> BOOL,
    _ >= _     : DECRAT x DECRAT -> BOOL,
    _ > _      : DECRAT x DECRAT -> BOOL,
    _ .eq. _   : DECRAT x DECRAT -> BOOL,
    _ .neq. _  : DECRAT x DECRAT -> BOOL
  variables
    nat, nat1, nat2 : NAT
    i : INT
    nz, nz1, nz2 : NZINT
    n, n1, n2 : NEGINT
    p, p1, p2 : POSINT
    dr, dr1, dr2 : DECRAT
    er : EXRAT
  equations
[D0]  intpart (dr) . decpart (dr) = dr
[D1]  intpart (i . nat) = i
[D2]  decpart (i . nat) = nat
[F0]  when 0 <= er = true and er < 1 = true then floor (er) = 0
[F1]  when 1 <= er = true
      then floor (er) = 1 + floor ((num (er) - denom (er)) /
                                   denom (er))
[F2]  when dec (0) <= er = true and er < 0 = true
      then floor (er) = dec (0)
[F3]  when er <= dec (0) = true
      then floor (er) = dec (0) + floor ((num (er) + denom (er)) /
```

```
                                                    denom (er))
[D2E1] when decpart (dr) .eq. 0 = true
       then dec2ex (dr) = intpart (dr)
[D2E2] when decpart (dr) .eq. 0 = false
      then dec2ex (dr) = dec2ex (10 * dr) / 10
[L1]  when p1 .eq. p2 = true
      then (p1 . nat1) < (p2 . nat2) = nat1 < nat2
[L2]  when p1 .eq. p2 = false
      then (p1 . nat1) < (p2 . nat2) = p1 < p2
[L3]  ((n1 . nat1) < (p2 . nat2)) = true
[L4]  ((p1 . nat1) < (n2 . nat2)) = false
[L5]  when p1 .eq. p2 = true
      then (n1 . nat1) < (n2 . nat2) = nat2 < nat1
[L6]  when p1 .eq. p2 = false
      then (n1 . nat1) < (n2 . nat2) = n1 < n2
[G]   (dr1 > dr2) = (dr2 < dr1)
[LE]  (dr1 <= dr2) = .not. (dr2 > dr1)
[GE]  (dr1 >= dr2) = .not. (dr1 < dr2)
[E]   (dr1 .eq. dr2) = ((dr1 <= dr2) .and. (dr2 <= dr1))
[NE]  dr1 .neq. dr2 = .not. (dr1 .eq. dr2)
end spec DecimalRationals

value spec Sets
  parameters
    Items begin
      sorts
        ITEM
      functions
        _ .eq. _ : ITEM x ITEM -> BOOL
    end Items
  imports
    Naturals,
    Booleans
  sorts
    SET
  functions
    empty_set    : SET,
    is_empty     : SET        -> BOOL,
    _ .in. _     : ITEM x SET -> BOOL,
    _ .nin. _    : ITEM x SET -> BOOL,
    _ .subset. _ : SET x SET  -> BOOL,
    _ .eq. _     : SET x SET  -> BOOL,
    insert       : ITEM x SET -> SET,
    delete       : ITEM x SET -> SET,
    _ + _        : SET x SET  -> SET,
    _ * _        : SET x SET  -> SET,
    _ - _        : SET x SET  -> SET,
    card         : SET        -> NAT
```

```
  variables
    s, s1, s2 : SET
    i, j : ITEM
  equations
[E1]   is_empty (empty_set) = true
[E2]   is_empty (insert (i, s)) = false
[IN1]  i .in. empty_set = false
[IN2]  when i .eq. j = true then i .in. insert (j, s) = true
[IN3]  when i .eq. j = false then i .in. insert (j, s) = i .in. s
[NIN]  i .nin. s = .not. (i .in. s)
[INS1] insert (i, insert (i, s)) = insert (i, s)
[INS2] insert (i, insert (j, s)) = insert (j, insert (i, s))
[DEL1] delete (i, empty_set) = empty_set
[DEL2] when i .eq. j = true
       then delete (i, insert (j, s)) = delete (i, s)
[DEL3] when i .eq. j = false
       then delete (i, insert (j, s)) = insert (j, delete (i, s))
[UN1]  empty_set + s = s
[UN2]  insert (i, s1) + s2 = s1 + insert (i, s2)
[INT1] empty_set * s = empty_set
[INT2] when i .in. s2 = true
       then insert (i, s1) * s2 = insert (i, s1 * s2)
[INT3] when i .in. s2 = false
       then insert (i, s1) * s2 = s1 * s2
[DIF1] s - empty_set = s
[DIF2] when i .in. s1 = true
       then s1 - insert (i, s2) = delete (i, s1) - s2
[DIF3] when i .in. s1 = false
       then s1 - insert (i, s2) = s1 - s2
[SUB]  when s1 * s2 = s1 then s1 .subset. s2 = true
[EQ]   s1 .eq. s2 = (s1 .subset. s2) .and. (s2 .subset. s1)
[C1]   card (empty_set) = 0
[C2]   when i .in. s = true
       then card (insert (i, s)) = card (s)
[C3]   when i .in. s = false
       then card (insert (i, s)) = inc (card (s))
end spec Sets

value spec ParNumbers
  parameters
    Numbers begin
      sorts
        NUMBER
      functions
        zero     : NUMBER,
        _ + _    : NUMBER x NUMBER -> NUMBER,
        _ .eq. _ : NUMBER x NUMBER -> BOOL,
        _ < _    : NUMBER x NUMBER -> BOOL,
```

```
        _ > _    : NUMBER x NUMBER -> BOOL
    end Numbers
  imports
    Booleans
end spec ParNumbers

value spec SetsOfNumbers
  imports
    Booleans,
    DecimalRationals,
    Sets using ParNumbers for Item
       binding [ITEM -> NUMBER, .eq. -> .eq.]
       renaming [SET -> NUMBER_SET]
  functions
    _ / _ : NUMBER x POSINT -> EXRAT,
    max   : NUMBER_SET      -> NUMBER,
    min   : NUMBER_SET      -> NUMBER,
    avg   : NUMBER_SET      -> EXRAT,
    sum   : NUMBER_SET      -> NUMBER
  variables
    numb : NUMBER
    numbs : NUMBER_SET
  equations
[MA1] max (empty_set) = zero
[MA2] when numb > max (numbs) = true
      then max (insert (numb, numbs)) = numb
[MA3] when numb > max (numbs) = false
      then max (insert (numb, numbs)) = max (numbs)
[MI1] min (empty_set) = zero
[MI2] when numb < min (numbs) = true
      then min (insert (numb, numbs)) = numb
[MI3] when numb < min (numbs) = false
      then min (insert (numb, numbs)) = min (numbs)
[A1]  avg (empty_set) = zero / 1
[A2]  avg (insert (numb, numbs)) =
      (numb / inc (card (numbs))) + avg (numbs) * card (numbs) /
        inc (card (numbs))
[S1]  sum (empty_set) = zero
[S2]  sum (insert (numb, numbs)) = numb + sum (numbs)
end spec SetsOfNumbers

value spec Bags
  parameters
    Item begin
      sorts
        ITEM
      functions
        _ .eq. _ : ITEM x ITEM -> BOOL
```

```
    end Item
  imports
    Naturals,
    Booleans
  sorts
    BAG
  functions
    empty_bag     : BAG,
    is_empty      : BAG         -> BOOL,
    _ .in. _      : ITEM x BAG -> BOOL,
    _ .nin. _     : ITEM x BAG -> BOOL,
    _ .subbag. _ : BAG x BAG  -> BOOL,
    _ .eq. _      : BAG x BAG  -> BOOL,
    insert        : ITEM x BAG -> BAG,
    delete        : ITEM x BAG -> BAG,
    _ + _         : BAG x BAG  -> BAG,
    _ * _         : BAG x BAG  -> BAG,
    _ - _         : BAG x BAG  -> BAG,
    card          : BAG         -> NAT
  variables
    b, b1, b2 : BAG
    i, j : ITEM
  equations
[E1]   is_empty (empty_bag) = true
[E2]   is_empty (insert (i, b)) = false
[IN1]  i .in. empty_bag = false
[IN2]  when i .eq. j = true then i .in. insert (j, b) = true
[IN3]  when i .eq. j = false then i .in. insert (j, b) = i .in. b
[NIN]  i .nin. b = .not. (i .in. b)
[INS2] insert (i, insert (j, b)) = insert (j, insert (i, b))
[DEL1] delete (i, empty_bag) = empty_bag
[DEL2] when i .eq. j = true
       then delete (i, insert (j, b)) = delete (i, b)
[DEL3] when i .eq. j = false
       then delete (i, insert (j, b)) = insert (j, delete (i, b))
[UN1]  empty_bag + b = b
[UN2]  insert (i, b1) + b2 = b1 + insert (i, b2)
[INT1] empty_bag * b = empty_bag
[INT2] when i .in. b2 = true
       then insert (i, b1) * b2 = insert (i, b1 * b2)
[INT3] when i .in. b2 = false
       then insert (i, b1) * b2 = b1 * b2
[DIF1] b - empty_bag = b
[DIF2] when i .in. b1 = true
       then b1 - insert (i, b2) = delete (i, b1) - b2
[DIF3] when i .in. b1 = false
       then b1 - insert (i, b2) = b1 - b2
[SUB]  when b1 * b2 = b1 then b1 .subbag. b2 = true
```

```
[EQ]   b1 .eq. b2 = (b1 .subbag. b2) .and. (b2 .subbag. b1)
[C1]   card (empty_bag) = 0
[C2]   card (insert (i, b)) = inc (card (b))
end spec Bags

value spec Strings
  parameters
    Item begin
      sorts
        ITEM
      functions
        _ .eq. _ : ITEM x ITEM -> BOOL,
        _ < _    : ITEM x ITEM -> BOOL
    end Item
  imports
    Booleans,
    Naturals
  sorts
    ITEM < STRING
  functions
    epsilon  : STRING,
    _ * _    : STRING x STRING -> STRING,
    length   : STRING           -> NAT,
    _ .eq. _ : STRING x STRING -> BOOL,
    _ < _    : STRING x STRING -> BOOL
  variables
    s, s1, s2 : STRING
    i, i1, i2 : ITEM
  equations
[A]   s1 * (s2 * i) = (s1 * s2) * i
[EP1] epsilon * s = s
[EP2] s * epsilon = s
[EQ1] (s1 * i1) .eq. (s2 * i2) = (s1 .eq. s2) .and. (i1 .eq. i2)
[EQ2] epsilon .eq. s = length (s) .eq. 0
[EQ3] s .eq. epsilon = length (s) .eq. 0
[L1]  length (epsilon) = 0
[L2]  length (i) = 1
[L3]  length (i * s) = inc (length (s))
[LE1] epsilon < epsilon = false
[LE2] epsilon < (i * s) = true
[LE3] (i * s) < epsilon = false
[LE4] when i1 .eq. i2 = true then (i1 * s1) < (i2 * s2) = s1 < s2
[LE5] when i1 .eq. i2 = false then (i1 * s1) < (i2 * s2) = i1 < i2
end spec Strings

value spec Alphabet
  imports
    Booleans,
```

```
    Naturals
  sorts
    ALPHA
  functions
    _ .eq. _ : ALPHA x ALPHA -> BOOL,
    _ < _    : ALPHA x ALPHA -> BOOL,
    ord      : ALPHA         -> NAT
  variables
    alpha1, alpha2 : ALPHA
  equations
[A57] alpha1 .eq. alpha2 = ord (alpha1) .eq. ord (alpha2)
[A58] alpha1 < alpha2 = ord (alpha1) < ord (alpha2)
end spec Alphabet

value spec Names
  imports
    Strings using Alphabet for Item
       binding [ITEM -> ALPHA, .eq. -> .eq., < -> <]
       renaming [STRING -> NAME]
end spec Names

value spec Dates
  imports
    Booleans,
    Naturals,
    Naturals using <identifier> for <identifier>
       renaming [NAT -> YEAR, inc -> next],
    Sets using Naturals for Item
       binding [ITEM -> NAT, .eq. -> .eq.]
  sorts
    MONTH, DAY, DATE < XDATE
  functions
    january          : MONTH,
    february         : MONTH,
    march            : MONTH,
    november         : MONTH,
    december         : MONTH,
    1st              : DAY,
    2nd              : DAY,
    28th             : DAY,
    30th             : DAY,
    31st             : DAY,
    thirty_one_days  : SET hidden,
    thirty_days      : SET hidden,
    date             : YEAR x MONTH x DAY -> DATE,
    year             : DATE               -> YEAR,
    month            : DATE               -> MONTH,
    day              : DATE               -> DAY,
```

```
    next            : DATE                -> DATE,
    next            : MONTH               -> MONTH,
    next            : DAY                 -> DAY,
    ord             : MONTH               -> NAT,
    ord             : DAY                 -> NAT,
    _ .before. _    : DATE x DATE         -> BOOL,
    future          : XDATE
    {-- Signature not regular, add declaration + : ZERO x ZERO -> ... }
  variables
    y, n : YEAR
    m : MONTH
    d : DAY
    date, dt1, dt2 : DATE
  equations
[I]   date (year (date), month (date), day (date)) = date
[Y]   year (date (y, m, d)) = y
[M]   month (date (y, m, d)) = m
[D]   day (date (y, m, d)) = d
[JAN] next (january) = february
[DEC] next (december) = january
[1st] next (1st) = 2nd
[30th] next (30th) = 31st
[OM1] ord (january) = 1
[OM12] ord (december) = inc (ord (november))
[OD1] ord (1st) = 1
[OD31] ord (31st) = inc (ord (30th))
[H1]  thirty_one_days = insert (1, insert (3, insert (5, insert (7,
            insert (8, insert (10, insert (12, empty_set)))))))
[H2]  thirty_days = insert (4, insert (6, insert (9,
                    insert (11, empty_set))))
[N1]  when (ord (m) .in. thirty_days) .and. ord (d) < 31 = true
      then next (date (y, m, d)) = date (y, m, next (d))
[N2]  when (ord (m) .nin. thirty_days) .or. ord (d) >= 31 = true
      then next (date (y, m, d)) = date (y, next (m), 1st)
[N3]  when ord (m) .eq. 2 = true and ord (d) < 28 = true
      then next (date (y, m, d)) = date (y, m, next (d))
[N4]  when (n * 4) .eq. y = true and (n * 100) .eq. y = false
      then next (date (y, february, 28th)) = date (y, march, 1st)
[N5]  when (n * 4) .eq. y = false
      then next (date (y, february, 30th)) = date (y, march, 1st)
[N]   when (n * 100) .eq. y = true
      then next (date (y, february, 30th)) = date (y, march, 1st)
[B]   dt1 .before. dt2 =
            (year (dt1) < year (dt2)) .or.
            ((year (dt1) .eq. year (dt2)) .and.
             (ord (month (dt1)) < ord (month (dt2)))) .or.
            ((year (dt1) .eq. year (dt2)) .and.
             (ord (month (dt1)) .eq. ord (month (dt2))) .and.
```

```
            (ord (day (dt1)) < ord (day (dt2))))
end spec Dates

value spec Times
  imports
    Naturals
       renaming [NAT -> HOUR],
    Naturals
       renaming [NAT -> MINUTE],
    Naturals
       renaming [NAT -> SECOND]
  sorts
    TIME < XTIME
  functions
    hours      : TIME                       -> HOUR,
    minutes    : TIME                       -> MINUTE,
    seconds    : TIME                       -> SECOND,
    time       : HOUR x MINUTE x SECOND -> TIME,
    inc_hour   : HOUR                       -> HOUR
    inc_minute : MINUTE                     -> MINUTE
    inc_second : SECOND                     -> SECOND
    later      : XTIME
  variables
    h : HOUR
    m : MINUTE
    s : SECOND
    t : TIME
  equations
[T1]  time (hours (t), minutes (t), seconds (t)) = t
[T2]  hours (time (h, m, s)) = h
[T3]  minutes (time (h, m, s)) = m
[T4]  seconds (time (h, m, s)) = s
[H1]  when h < 23 = true then inc_hour(h) = inc(h)
[H2]  inc_hour (23) = 0
[M1]  when m < 59 = true then inc_minute(h) = inc(h)
[M2]  inc_minute (59) = 0
[S1]  when s < 59 = true then inc_second(h) = inc(h)
[S2]  inc_second (59) = 0
end spec Times

value spec Money
  imports
    DecimalRationals
  sorts
    MONEY < DECRAT
  variables
    dr : DECRAT
  sort constraints
```

```
[SC1] when intpart (100 * dr) .eq. (100 * dr) = true then dr : MONEY
end spec Money
```

Bibliography

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *The First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, december 1989.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[3] J.A. Bergstra. A process creation mechanism in process algebra. In J.C.M. Baeten, editor, *Applications of Process Algebra (CWI Monograph 8)*, pages 81–88. North-Holland, 1989.

[4] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press/Addison Wesley, 1989.

[5] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* B.G. Teubner, 1989.

[6] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* Springer, 1985. EATCS Monographs on Theoretical Computer Science, Vol. 6.

[7] L.T.F. Gamut. *Logica, Taal en Betekenis 2: Intensionele Logica en Logische Grammatica.* Het Spectrum, 1982. L.T.F. Gamut is a pseudonym for J.F.A.K. van Benthem, J. Groenendijk, D. de Jongh, M. Stokhof, and H. Verkuyl.

[8] R. J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions.* PhD thesis, Vrije Universiteit/Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[9] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025, U.S.A., 1988.

[10] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Given as lecture at Seminar on Types, Carnegie-Mellon University, June 1983; many draft versions exist.

[11] E. Hamoen. An order-sorted rewrite module: Start-up report. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, April 1991.

[12] D. Harel. On visual formalisms. *Communications of the ACM*, 31:514–530, 1988.

[13] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 789–840. Elsevier Science Publishers, 1990.

[14] B.J. MacLennan. Values and objects in programming languages. *Sigplan Notices*, 17(12):70–79, December 1982.

[15] J.-J.Ch. Meyer and R.J. Wieringa. Actor-oriented system specification with dynamic logic. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 2, pages 337–357. Springer, 1991. Lecture Notes in Computer Science 494.

[16] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In M. Nivat and H. Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 297–367. Academic Press, 1989.

[17] P.A. Spruit. Two finite-graph models for basic process algebra. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1990.

[18] P.A. Spruit. Finite-graph models for Process Algebra with parallel composition. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, In preparation.

[19] P.A. Spruit and C.J.A. Duivenvoorde. A syntax- and semantics-directed editor for conceptual model specifications. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, April 1990.

[20] P.A. Spruit and R.J. Wieringa. Some finite-graph models for process algebra. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, February 1991. To be published, CONCUR'91.

[21] H.R. Walters. Hybrid implementations of algebraic specifications. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic programming*, pages 40–54. Springer, 1990. Lecture Notes in Computer Science 463.

[22] R.J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.

[23] R.J. Wieringa. Equational specification of dynamic objects. In *IFIP WG2.8 Working Conference on Database Semantics.* North-Holland, 1990. To be published.

[24] R.J. Wieringa. An integrated specification of values, objects and processes for object-oriented models. In J. Göers and A. Heuer, editors, *Proceedings, 2nd International Workshop on Foundations of Models and Languages for Data and Objects*, pages 199–208. Institut für Informatik, Technische Universität Clausthal, Postfach 1253, 3392 Clausthal-Zellerfeld, Germany, September 1990.

[25] R.J. Wieringa. Object-oriented analysis, structured analysis, and Jackson system development. Technical report, 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.

[26] R.J. Wieringa. A formalization of objects using equational dynamic logic. Technical report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, January 1991. Submitted for publication.

[27] R.J. Wieringa. Steps towards a method for the formal modeling of dynamic objects. *Data and Knowledge Engineering*, To be published.

[28] R.J. Wieringa and J.-J.Ch. Meyer. Actor-oriented specification of dynamic and deontic integrity constraints. In B. Talheim, J. Demetrovics, and H.-D. Gerhardt, editors, *3rd Symposium om Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS 91)*, pages 89–103. Springer, 1991. Lecture Notes in Computer Science 495.

[29] R.J. Wieringa and R.P. Van De Riet. Algebraic specification of object dynamics in knowledge base domains. In R.A. Meersman, Zhongshi Shi, and Chen-Ho Kung, editors, *Artificial Intelligence in Databases and Information Systems (DS-3)*, pages 411–436. North-Holland, 1990.

Contents

1 Introduction **1**
- 1.1 The purpose of CMSL . . . 1
- 1.2 Objects and values . . . 2
- 1.3 Objects and values in databases and programming languages . . . 4
- 1.4 Objects and values in the CM . . . 5

2 The value specification language VSL **7**
- 2.1 Specification . . . 7
  - 2.1.1 Introduction . . . 7
  - 2.1.2 Research problems . . . 8
- 2.2 Interaction . . . 9
  - 2.2.1 Introduction . . . 9
  - 2.2.2 Research problems . . . 10

3 The object specification language OSL **11**
- 3.1 Attributes . . . 11
  - 3.1.1 Specification . . . 11
  - 3.1.2 Research problems . . . 15
  - 3.1.3 Implementation . . . 16
  - 3.1.4 Interaction . . . 17
  - 3.1.5 Research problems . . . 18
- 3.2 Events . . . 18
  - 3.2.1 Event theory specification . . . 18
  - 3.2.2 Event specification . . . 19
  - 3.2.3 Research problems . . . 22
  - 3.2.4 Implementation . . . 23
  - 3.2.5 Research problems . . . 24
- 3.3 Life cycles . . . 24
  - 3.3.1 Process theory specification . . . 24
  - 3.3.2 Research problems . . . 27
  - 3.3.3 Life cycle definition . . . 27
  - 3.3.4 Research problems . . . 28
  - 3.3.5 Implementation . . . 29

4 The conceptual model specification language (CMSL) **30**
- 4.1 Specification . . . 30
- 4.2 Implementation . . . 30

A BNF grammar of CMSL **31**
A.1 VSL 31
A.2 OSL 33
A.3 CMSL 34

B Example ADT specifications **35**