Equational Specification of Dynamic Objects

Roel Wieringa

Department of Mathematics and Computer Science
Free University
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands
uucp: roelw@cs.vu.nl

ABSTRACT

An equational language to specify object-oriented conceptual models is defined. Objects are considered to be characterized by a unique object identifier and have static and dynamic structure. Examples of static structure are classification, aggregation, generalization and grouping, examples of dynamic structure are events, processes, local (intra-object) and global (inter-object) and communication. The language, called CMSL, has a declarative (algebraic) semantics, which is divided into two. The part of CMSL that can be used to specify static structures has an initial algebra semantics, in which the data elements are object versions. The part dealing with process has a larger algebra as semantics; in this paper we use an algebra of graphs modulo bisimulation equivalence. About both models can be reasoned using standard equational logic. Apart from the combination of static and dynamic features of objects in an algebraic framework, and the joint specification of this in an equational language, other features of CMSL are the specification of roles (classes of which an object may become a member or cease to be a member) and the use of structured identifiers to represent higher-order objects.

1. Introduction

The aim of conceptual modeling is to specify an explicit model of a universe of discourse (UoD). The conceptual model (CM) is a mathematical structure which is an abstract representation of the UoD. In general, there is also a normative relation between the CM and the UoD, but we ignore this in this paper [34]. We thus regard the relation between the CM and the UoD as analogous to the relation between an abstract physical model of reality and reality itself (e.g. between an ideal gas and a real gas).

The specification of the CM (CMS) and the CMS is a theory in a formal language, i.e. it is a set of sentences. The relation between the CMS and the CM is that the CM is a (unique or preferred) denotational model of the CMS. This is so, regardless of the relation between the CM and the UoD (normative and/or descriptive) [35].

The aim of this paper is to define an equational language, called CMSL (Conceptual Model Specification Language), in which one can specify object-oriented models. The language is designed specifically with a view to specifying the structures usually found in CM's, such as classification, generalization, and complex objects. An important element of the language is its facility to specify object dynamics in addition to object statics. Thus, events, event preconditions, processes and communication can be specified. This paper is a brief overview of [37], where a detailed analysis of the CM structures and a comparison with other modeling approaches can be found. We concentrate on the syntax of the language and discuss the semantics only by means of examples. A companion paper

[38] discusses the formal semantics of CMSL.

Other approaches to formal object specification in a database context can be found in work done by Goguen & Meseguer on the FOOPS language [19, 22], and in the work done by a number of researchers in the ISCORE project. The FOOPS language is an extension of OBJ [20] with the possibility to specify objects. We start from the much simpler equational language ASF developed by Bergstra, Heering and Klint [9], and extend it with the possibility to partially order sorts and to specify sort constraints, to bring it closer to OBJ3. Like is done in OBJ, we assume an initial semantics for value specifications [21], but unlike FOOPS, we also give an initial semantics for attribute- and event specifications. The published version of FOOPS has no facility for the specification of life cycles, which is one of the major features of CMSL.

Within the ISCORE project, work is done on the object specification language OBLOG [10, 31], which has roughly the same capabilities as CMSL, but does not (yet) allow the specification of roles that objects may play. Role specification is briefly discussed in section 4.1 below. OBLOG is not a purely equational language with an algebraic semantics, but has a more general semantics based on categorical constructions [11, 12]. This semantics is not bound to OBLOG but is used to study general issues like object generalization and aggregation in a unified framework. Our approach in CMSL is more simplistic and tries to get as far as possible within the initial algebra framework, extending it where necessary with work done in process algebra [5]. Fiadeiro et al. [15, 14] extend this framework with the capability to specify normative behavior of objects, and to prove properties of objects in temporal logic. We ignore deontic aspects in this paper, but plan to add them later in the context of dynamic logic [26, 35].

In section 2 of this paper we introduce some of the static structures specifyable in CMSL, and we give some examples of local and global integrity constraints. A very brief overview of the formal semantics of this part of the language is given, which is based on initial algebras.

In section 3, dynamic structures are added. These are events, which may change attributes values, and processes, which consist of events composed with operators for sequence, choice, parallel composition, and communication. The initial semantics of objects is extended with event semantics. Processes have a different semantics, which is not initial.

In section 4 we list a few of the other capabilities of CMSL, and in section 5 we summarize the paper and give the direction of our current research. A detailed presentation of the material summarized in this paper can be found in [37].

2. Static structures

Most structures found in semantic data models are static. In this section, we treat classification, aggregation, grouping of objects, and the taxonomic structure of objects.

2.1. Value specifications

CMSL consists of a language to specify abstract data types (VSL, value specification language), objects (OSL, object specification language), and processes (PSL, process specification language). VSL is based on the equational specification language ASF (Amsterdam Specification Formalism), developed by Bergstra, Heering, and Klint [9], extended with the possibility to partially order sorts and to specify sort constraints [16, 19, 18]. The sorts specified in VSL are called *value sorts.* In CMSL specifications, data types like naturals, integers, rationals, money, date and time, and parametrized data types like sets, bags and strings are standard, the information analyst can add his or her own data type specifications. Example value specifications are given in [37].

Object identifiers

An object specification can import value specifications or other object specifications, and designates an arbitrary set I of value sorts as *identifier sorts.* Data elements of these types will serve as object identifiers. I must be downward closed with respect to the ordering $\leq$ on sorts, i.e. if $s \in I$ and $s' \leq s$, then $s' \in I$. A very basic object specification example is

```
object spec Persons1
    identifier sorts
        PERSON
    functions
        p0  : PERSON
        new : PERSON -> PERSON
end spec Persons1
```

In CMSL, `PERSON`, `p0`, and `new` can be made part of a value specification which is then imported in `Persons1`, whose only job would then be to declare `PERSON` to be an identifier sort. It is often shorter to write the declarations of functions for the identifier sort in the object specification itself. Treating identifier sorts as any other value sort, and using an initial algebra semantics, `PERSON` denotes the type of data elements generated by `p0` and `new`,

```
    p0, new(p0), new(new(p0)), ...
```

Using this semantics for the moment, `Persons1` specifies a CM that only represents the fact that there are infinitely many different persistent entities of sort `Person`. Nothing else is represented. Note that, in addition to numerical difference and persistence of objects, identifiers carry a third piece of information, viz. that they are elements of an identifier sort. (In general, identifier sorts form a poset and an identifier is element of more than one identifier sort.) Roughly, identifier sorts are the classes as which we *classify* all possible entities in the UoD, each possible object is an instance of at least one such class. Note that this kind of classification is *essential* in that set membership is essential. If `p` is a person, then `p` is essentially a person, because we cannot move elements of sets to other sets. If we remove an element of `PERSON` then the result is a set different from `PERSON` because the identity of sets is determined by their members. In section 4.1, we discuss a kind of classifying objects which is less rigid.

2.2. Attributes

Contingent properties of objects are represented by attributes, which in CMSL, must be unary functions with an identifier sort as argument sort and any value sort as result sort. This includes identifier sorts as possible result sorts.

```
object spec Addresses1
    import
        ZipCodes, Strings
    identifier sorts
        ADDRESS
    functions
        a0  : ADDRESS
        new : ADDRESS -> ADDRESS
    attributes
        zip  : ADDRESS -> ZIP
        city : ADDRESS -> STRING
end spec Addresses1
```

`Addresses1` imports two value specifications, which declare and specify types `ZIP` and `STRING` (not shown here). It declares denumerably many different address identifiers, generated by `a0` and `new`, each of which have two attributes, `zip` and `city`. For each address `a`, `zip(a)` and `city(a)` are syntactically well-formed terms.

To interpret attributes, we take as CM a set of possible worlds (i.e. a Kripke structure). Each possible world of the CM is an algebra which interprets the attribute names as functions of the appropriate arities. Thus, in each possible world, `zip(a)` delivers the zip code of `a` in that world. In the next section, we look closer at the structure of the CM. Here, we add some syntactic remarks.

First, a CM specification is basically an empty shell which imports value specifications, object specifications, and possibly other CM specifications. In each CM specification, each attribute name can be declared at most once. This is analogous to the *universal relation scheme assumption* (URSA), which says that each attribute name denotes the same class of entities in every context [24, 25, 32]. The difference with that assumption in relational theory is that we distinguish attributes (e.g. `city`) from their value ranges (e.g. `STRING`). This gives us two forms of the URSA. One is that every attribute range name denotes the same data type in every context. Thus, there is only one data type denoted by `ADDRESS`. This amounts to the requirement that `ADDRESS` is specified only once, which is a strict form of the rule that every occurrence of a name should be traceable to a unique declaration [9]. The second form of the URSA is actually mentioned in [24] and is called the *unique role assumption.* It says that each attribute name denotes the same role every time it occurs. In *CMSL*, we reserve the word ‘‘role’’ for another purpose and the unique role assumption boils down to the requirement that each attribute be declared at most once.

2.3. The version algebra, possible worlds, and objects

It is interesting to see what happens if we interpret an attribute name in an initial algebra. If `zip` would be interpret initially, then the data type `ZIP` would be extended with closed terms of the form `zip(a)` (modulo equality), with `a` a closed term of sort `ADDRESS`. This is not what we want. What we want instead is to interpret attributes in *possible worlds,* where a possible world is defined as an algebra of an object specification which contains the value algebra as reduct. Skipping the formal definitions (see [13] for a definition of reducts and [37, 38] for a definition of possible worlds of a CM), this means the following. Each object specification assumes some value specifications, for example of Booleans or Naturals. An object specification adds attribute declarations, and we now require the attributes not to extend the value types, i.e. they must be interpreted as functions on the value sorts as specified in the value specifications. Thus, in each possible world, `zip(a)` returns one of the zip codes specified in the value specification of `ZIP`, as expected. In different possible worlds it may return different zip codes for the same argument `a`. The possible worlds of an object specification form a simple S5 Kripke structure. The set of possible worlds is called *PW*, with typical element *w*.

Having interpreted an object specification in a Kripke structure, it would be reasonable to define a modal equational inference system in which we can reason about attribute values in possible worlds. We choose another, simpler way to reason about attributes, by using equational deduction without modal inference rules. The idea is to transform an object specification $Spec_{Object}$ into a value specification $V(Spec_{Object})$, which has an initial algebra called the *version algebra* of the object specification. With respect to this initial algebra, we can use classical equational deduction, and, by initiality, structural induction on closed terms. $V(Spec_{Object})$ is called a *specification expansion,* and *V* stands for "Version specification." The version algebra has *object versions* as data elements, which have the form

```
<a, <zip: z, city: s>>.
```

Attributes are projection functions on these object versions. We give the expansion of `Addresses1` and then explain its elements.

```
value spec V(Addresses1)
    import
        ZipCodes, Strings
    sorts
        ADDRESS, ADM-ADDRESS-VERSION < ADDRESS-VERSION
    functions
        a0     : ADDRESS
        new    : ADDRESS         -> ADDRESS
        <_, _> : ADDRESS x <zip: ZIP, city: STRING>
                                 -> ADDRESS-VERSION
        id     : ADDRESS-VERSION -> ADDRESS
        zip    : ADDRESS-VERSION -> ZIP
        city   : ADDRESS-VERSION -> STRING
    variables
        av : ADDRESS-VERSION
        a  : ADDRESS
        z  : ZIP
        s  : STRING
    equations
[E0]    <id(av), <zip(av), city(av)>> = av
[E1]    id(<a, <zip: z, city: s>> = a
[E2]    zip(<a, <zip: z, city: s>> = z
[E3]    city(<a, <zip: z, city: s>> = s
end spec V(Addresses1)
```

For each identifier sort *s*, the expanded specification contains the declaration $adm-s-version \leq s-version$ (we write `<` instead of ≤ in CMSL). Informally, `ADDRESS-VERSION` is the sort of all possible address versions, and `ADM-ADDRESS-VERSIONS` is the sort of all *admissible* address versions, which are defined as those versions that satisfy the integrity constraints for addresses. Integrity constraints are defined below. In general, the subsort ordering on sort names is interpreted a the subset ordering on sorts. Since there are no integrity constraints for address versions, all address versions are admissible address versions.

Version sorts are generated by the two-place operator `<_, _>`, called the *version generator*. The labeled tuple `<zip: z, city: c>>` in address versions is called an *attribute vector* and contains pairs $a : s$ for each attribute *a* applicable to `ADDRESS`, where *s* is the result sort of *a* (this is unique because *a* is declared only once). The *applicable* attributes are the attributes defined for `ADDRESS` and for all its supersorts; since there are no supersorts, `zip` and `city` are the only applicable attributes. `ADDRESS-VERSION` is the set of all possible address versions. The equations `[E0-3]` define the attributes as projection functions on `ADDRESS-VERSION`. The version algebra can be visualized as shown in table 1.

`<a0, <zip: z10, city: s10>>`	`<a1, <zip: z11, city: s11>>`	…
`<a0, <zip: z20, city: s20>>`	`<a1, <zip: z21, city: s21>>`	…
…	…	…

Table 1. The address version algebra.

All entries in the table are data elements of sort `ADDRESS-VERSION`. The table is organized suggestively, such that each column contains all versions identified by a single identifier, and each row contains exactly one version for each address identifier. Although not represented clearly in the table, different versions in one row may have the same attribute values. The table allows us to visualize our formalization of the concepts of object and world.

An *object* is, roughly, a column in table 1. More formally, an address object is a subalgebra of the version algebra of `Addresses1`. There is thus a 1-1 correspondence between identifiers and objects in the sense that each identifier identifies exactly one object. All objects of the same sort are isomorphic, because they differ only in their identifier. This formalization captures the idea of *locality* implicit in the concept of objects in that each object encapsulates attribute values, which thereby are local to the object.

A *world* is, roughly, a row in table 1. More formally, a world is a function from object identifiers to attribute vectors. This corresponds to the idea of an assignment to variables in denotational semantics, which is a function assigning each variable exactly one value. The difference is that in CM's, the value has a systematic internal structure (it is a labeled tuple of values), and that the identifier is part of the model of the specification, not part of the specification itself. This allows us to use infinitely many different identifiers and to give identifiers an internal structure, as illustrated below. More on this can be found in [37, 38]

Earlier, we defined the CM of an object specification as a set of possible worlds, each of which contains the intended value algebra as reduct. Here, we defined a world as a function from identifiers to attribute vectors. It can be proven that the two concepts are equivalent [37].

Object aggregation

Attribute ranges can be any value sort, including identifier sorts. This gives us object aggregation. An example is:

```
object spec Persons2
    import
        Naturals, Addresses1
    identifier sorts
        PERSON
    functions
        p0  : PERSON
        new : PERSON -> PERSON
    attributes
        age     : PERSON -> NAT
        address : PERSON -> ADDRESS
end spec Persons2
```

In a single world of the CM of this specification, we may have object versions like

```
<p0, <age: 13, address: a0>>,
<a0, <zip: 12345, city: "Amsterdam">>.
```

The arities of the attributes can be shown in a graph called the *aggregation graph* of `Persons2`, see figure 1. The aggregation graph may be cyclic, take for example the attribute `spouse : PERSON -> PERSON`.

Integrity constraints

Attributes can be subject to *integrity constraints*, which are either *local* or *global*. Syntactically, all attributes in a local constraint are applied to the same identifier variable. Semantically, a local constraint defines the admissible versions of single objects, and global constraints define the admissible worlds consisting of at least two objects. For example,

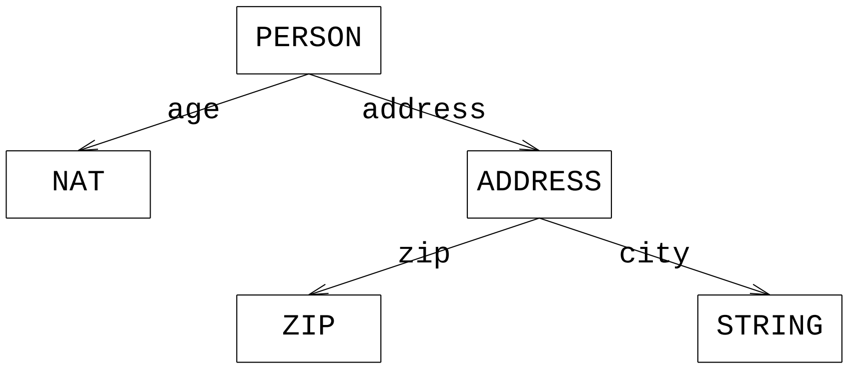

Figure 1.

```
object spec Persons3
    import
        Naturals, Addresses1
    identifier sorts
        PERSON
    functions
        p0  : PERSON
        new : PERSON -> PERSON
    attributes
        age     : PERSON -> NAT
        address : PERSON -> ADDRESS
        spouse  : PERSON -> PERSON
    variables
        p : PERSON
    local attribute constraints
[LAC1]  spouse(p) eq p = false
[LAC2]  age(p) < 150 = true
    global attribute constraints
[GAC1]  spouse(spouse(p)) = p
end spec Persons3
```

`[LAC1]` says that a person cannot be married to him- her herself. This is a local constraint, because it can be said to be true or false of a single person version. `[GAC1]` says that the universe represented by the specified model is monogamous. This is a global constraint, because it is true or false of sets of two person versions.

Global constraints are ignored in the expansion of the person specification to a version algebra specification, but local constraints play in important role. They are expanded to *sort constraints* for the sort `ADM-PERSON-VERSION`. Roughly, these say when a term of sort `PERSON-VERSION` is of sort `ADM-PERSON-VERSION`. (Sort constraints are introduced in [16] and used in the FOOPS language19]. A more thorough treatment is promised in a sequel to [21].) In the version algebra, `ADM-PERSON-VERSION` is the largest set of (equivalence classes of) closed terms satisfying all local attribute constraints for persons.

Global constraints have no meaning in terms of object versions but in terms of possible worlds. A possible world satisfying all local and global attribute constraints is called *admissible*. The set of admissible worlds is called *AW*.

Structured identifiers

Next to classification and aggregation of objects, there are other structuring principles known from semantic modeling, such as association or *grouping*, and Cartesian products. We represent these structures by means of *structured identifiers*. $s \in I$ is a *primitive identifier sort* iff its elements are generated from elements of sort *s*, otherwise it is a *structured identifier sort*. PERSON and ADDRESS are examples of primitive identifier sorts. In CMSL specifications, we assume a supersort PRIM of all identifier sorts. We then get the following specification.

```
value spec PrimitiveIdentifiers
    import
        Booleans
    sorts
        PRIM
    functions
        new : PRIM        -> PRIM
        eq  : PRIM x PRIM -> BOOL
    ; equations defining eq to be the equality predicate omitted
end spec PrimitiveIdentifiers

object spec Addresses2
    import
        PrimitiveIdentifiers, ZipCodes, Strings
    identifier sorts
        ADDRESS specializing PRIM
    functions
        a0  : ADDRESS
    attributes
        zip  : ADDRESS -> ZIP
        city : ADDRESS -> STRING
end spec Addresses1

object spec Persons4
    import
        Addresses2, Naturals, PrimitiveIdentifiers
    identifier sorts
        PERSON specializing PRIM
    functions
        p0  : PERSON
    attributes
        ; the rest is the same as in Persons3
end spec Persons4
```

There are no constants declared in PrimitiveIdentifiers, so in the initial algebra of that specification, PRIM is empty. In the possible worlds of the model specified by Addresses2, on the other hand, PRIM contains elements

```
a0, new(a0), ...,
```

and in the possible worlds of the model specified by Persons4, PRIM contains elements

```
    a0, p0, new(a0), new(p0), ....
```

Below we show that in this way we can specify any desired taxonomic structure. Note that the `new` function should have been declared for every subsort *s* of `PRIM`, otherwise *s* is not generated properly. To save space we omit this declaration from the examples.

`ADDRESS` and `PERSON` are subsorts of `PRIM` as indicated by the keyword **specializing**. This is synonymous to `<`, but is meant to draw attention of the specifier that identifier sorts may extend their supersort. An example of a structured identifier sort is:

```
object spec HigherOrderObjects1
    import
        Persons4, ExactRationals,
        Sets using Persons4 for Items
             binding [ITEM -> PERSON,
                      eq   -> eq]
             renaming [SET -> PERSONS]
    identifier sorts
        PERSONS
    attributes
        children : PERSON  -> PERSONS
        avg-age  : PERSONS -> EXRAT
end spec HigherOrderObjects1
```

`ExactRationals` is a specification of rationals in exact notation (e.g. `3 / 4`, see [17, 20, 37]), whose sort of interest is `EXRAT`. `Sets` is specification of sets parametrized by `Items`. This contains sorts `ITEM` and `SET` [37]. `ITEM` is bound to `PERSON` and `SET` is renamed to `PERSONS`. `PERSONS` is a structured identifier sort because it is generated from `PERSON` elements, using the generators `empty-set` and `insert` (declared in `Sets`).

Next, we declare attributes using `PERSONS`. `children` can be regarded as a *multi-valued* attribute. Person versions now have the form

```
    <p, <age: n, address: a, children: {p1, ..., pn}>>.
```

Note that `{p1, ..., pn}` is a single value of type `PERSONS`, and that al set operators defined in `Sets` apply to it. Using sets to represent multi-valued attributes allows us to use the value `empty-set` to indicate positive knowledge that a person has no children.

`avg-age` is called a *higher-order attribute* because it has a structured identifier sort as argument sort. The model can now contain object versions of the form

```
    <{p1, ..., pn}, <avg-age: r>>
```

for `r` of type `EXRAT` (which is declared in `ExactRationals`). Due to the equations in `Sets`, this object version has the same identifier as `{p1} + {p2, ..., pn}`, but a different identifier from `{p0} + {p1, ..., pn}` (assuming all `pi` are different).

To exclude inadmissible object versions, we should add some constraints. The following local attribute constraint for `children` prevents a person from being his or her own child,

```
    p in children(p) = false.
```

The constraints for `avg-age` are more complex. Let `{{...}}` denote a multiset (or bag) and suppose that we have a function `avg : BAG-OF-NAT -> EXRAT` which computes the average of a bag of naturals. Then our intention is that

```
      avg-age({p1, ..., pn}) = avg({{age(p1), ..., age(pn)}}).
```

This is done in the following specification.

```
object spec HigherOrderObjects2
    import
        Persons4, ExactRationals, Naturals,
        BagsOfNaturals,
        Sets using Persons4 for Items
            binding [ITEM -> PERSON,
                     eq   -> eq]
            renaming [SET -> PERSONS]
    identifier sorts
        PERSONS
    attributes
        children : PERSON  -> PERSONS
        avg-age  : PERSONS -> EXRAT
        ages     : PERSONS -> BAG-OF-NAT
    variables
        p  : PERSON
        pp : PERSONS
    local attribute constraints
[LAC1]  p in children(p) = false
[LAC2]  avg-age(pp) = avg(ages(pp))
    global attribute constraints
[GAC1]  ages(empty-set) = empty-bag
[GAC2]  ages(insert(p, pp)) = insert(age(p), ages(pp))
end spec HigherOrderObjects2
```

We assume a specification of bags of naturals, with sort of interest `BAG-OF-NAT` and a function `avg : BAG-OF-NAT -> EXRAT` which computes the average of a bag of naturals. Note that `insert` occurs overloaded in `[GAC2]`. Its second argument is a set at the left-hand side and a bag at the right-hand side.

In CMSL, we can map any attribute `a : s1 -> s2` over `set-of-s1` by writing `map-a`; the required declarations and equations are then implicitly added. In the same way, we can also specify inverse attributes (e.g. `inv-age` for all persons having a certain age). `map-` and `inv-` are really constructors which accept an attribute as argument and yield a higher-order attribute as result. Goguen & Meseguer [17] use a similar mechanism to define higher-order attributes in a first-order language, EQLOG. OBJ3 also contains this mechanism [20].

We now have seen examples of every piece of information that an object identifier can carry. Each identifier contains essential classification information, expresses numerical difference from different objects and persistence of an object through change. Finally, structured identifiers contain essential structural information. Note that identifiers in programming languages contain the same four pieces of information. An identifier used as the name of a variable can be used to distinguish the variable from others that have the same value, to indicate persistence of the variable through change of value, to bind the variable to a type, and to define structured identifiers (e.g. records and arrays) from more simple types of identifiers.

Taxonomic structures

Next to classification, aggregation, and grouping of objects, generalization is an important structuring technique in semantic databases. The taxonomy of objects is given by the ordering on identifier sorts and by the way those sorts are generated. The following example illustrates this. We only give identifier sort and function declarations.

```
object spec Vehicles
    import
        PrimitiveIdentifiers
    identifier sorts
        VEHICLE specializing PRIM
        {MOTOR-VEHICLE, AIR-VEHICLE} specializing VEHICLE
        MOTOR-AIR-VEHICLE specializing {MOTOR-VEHICLE, AIR-VEHICLE}
    functions
        mv0  : MOTOR-VEHICLE
        av0  : AIR-VEHICLE
        mav0 : MOTOR-AIR-VEHICLE
end spec Vehicles
```

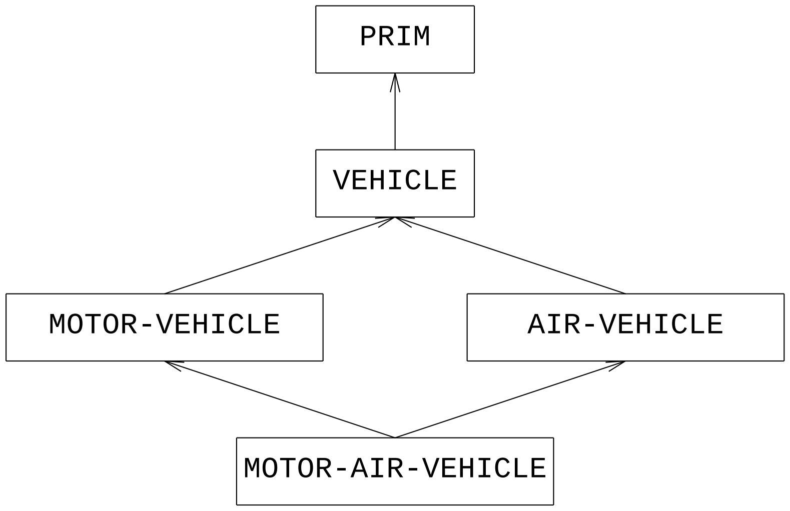

Figure 2.

If $s_1, s_2 \in I$ and $s_1 \le s_2$, then s_1 is called a *specialization* of s_2, and s_2 is called a *generalization* of s_1. Multiple specializations or generalizations can be indicated by brackets. By the placement of constants in some sorts but not in others, we specify which identifier sorts are disjoint and whether an identifier sort is exhausted by its subsorts. We write $[\![s]\!]$ for the identifier sort denoted by $s \in I$. Then we have

$$
\begin{aligned}
&[\![\texttt{MOTOR-AIR-VEHICLE}]\!] = \{\texttt{mav0}, \texttt{new(mav0)}, \ldots\},\\
&[\![\texttt{MOTOR-VEHICLE}]\!] = [\![\texttt{MOTOR-AIR-VEHICLE}]\!] \cup \{\texttt{mv0}, \texttt{new(mv0)}, \ldots\},\\
&[\![\texttt{AIR-VEHICLE}]\!] = [\![\texttt{MOTOR-AIR-VEHICLE}]\!] \cup \{\texttt{av0}, \texttt{new(av0)}, \ldots\},\\
&[\![\texttt{PRIM}]\!] = [\![\texttt{VEHICLE}]\!] = [\![\texttt{MOTOR-VEHICLE}]\!] \cup [\![\texttt{AIR-VEHICLE}]\!].
\end{aligned}
$$

We can illustrate the subsort relation between identifier sort in a *generalization graph*, in which the arrows point in the direction of the larger sort in the subsort ordering (so one can think of them as injection functions), see figure 2. The generalization graph is acyclic because $\le$ is a partial ordering, but it may be disconnected. For example, if we would add a structured identifier sort like `VEHICLES`, it would not be connected to the other nodes in figure 2.

3. Dynamic structures

The real world contains more than only static structures. Objects have behavior, by which we mean that they can execute *events* and that the life of an object is a *process* composed of these events.

3.1. Events

In the version algebra, an event is a function which maps an object version to an object version, keeping the identifier invariant. In terms of table 1, an event moves an object version up or down a column. In CMSL, an event is declared as a function

$$e: s_1 \times \ldots \times s_n \longrightarrow s_1,$$

where s_1 is an identifier sort and $n \geq 1$. s_1 is called the *subject sort* and $s_2, \ldots, s_n$ are called the *parameter sorts* of the event. In an application $e(x_1, \ldots, x_n)$, x_1 is called the *subject* and $x_2, \ldots, x_n$ the *parameters* of the application. Instead of viewing an event as a function on object versions, we may also see it as a function on possible worlds, which, when applied to an object version in a world, returns a world containing the updated object version. The intention is that e is applied in a world and x_1 identifies the object version in the world to which the event is applied.

We give a brief sketch of the semantics of events as functions on object versions. Consider the following example, which extends `Persons4` with an **events** section containing event declarations, and a **local event constraints section** with equations defining the effect of the events.

```
object spec Persons5
    ; all other sections the same as Persons4
    events
        inc-age        : PERSON             -> PERSON
        change-address : PERSON x ADDRESS -> PERSON
        get-spouse     : PERSON x PERSON  -> PERSON
    variables
        p, p1, p2 : PERSON
        a : ADDRESS
    local event constraints
[LEC1]  age(inc-age(p)) = age(p) + 1
[LEC2]  address(change-address(p, a)) = a
[LEC3]  spouse(get-spouse(p1, p2)) = p2
end spec Persons5
```

In a CM specification, each event name can be declared only once. This parallels the requirement that attribute name declarations are unique.

In the expansion of the object specification to the version algebra specification, the subject sort is replaced by the corresponding version sort. So `V(Persons5)` contains declarations like

```
     inc-age        : PERSON-VERSION             -> PERSON-VERSION
     change-address : PERSON-VERSION x ADDRESS -> PERSON-VERSION
     get-spouse     : PERSON-VERSION x PERSON  -> PERSON-VERSION.
```

To define the effect of these functions, the local event constraints are transformed into equations for these functions. For example, `[LEC1]` is transformed into

```
age(inc-age(pv)) = age(pv) + 1,
```

where `pv` is a variable of sort `PERSON-VERSION`. In general, the subject argument is replaced by the corresponding version variable. This gives us three equations, which allow us, for example, to reduce `address(change-address(pv, a))` to `a`. However, the equations do not define the effect of the events in the version algebra sufficiently. For example, `age(change-address(pv, a))` cannot be reduced to a value in `NAT`, because there are no equations for it. We therefore use a *local frame assumption* during specification expansion, which says that if the value of an attribute *a* after an application of event *e* cannot be reduced to a data element of the range of *a*, then *e* does not change the value of *a*. We then add a *frame equation* like

```
age(change-address(pv, a)) = age(pv).
```

If we would not add the frame equations, then `NAT` would be extended with closed terms like `age(change-address(pv, a))` for closed terms `pv` of sort `PERSON-VERSION` and closed terms `a` of sort `ADDRESS`. Note that the decision when to add a frame equation is in general undecidable, because it requires an answer to the question whether a certain reduction can take place or not.

We have built in a *global frame assumption* by defining events as functions on object versions. This means that an event only changes the attributes of its subject and of no other object. Events are thus local. Another way of saying this is that events are encapsulated in an object. By implication, because attributes can only be changed by events, attributes are encapsulated in an object. Because the word "encapsulation" is used in a different way in process algebra, we use the word "local" instead: event applications and attribute values are *local* to an object. We again see that the object identifier is a *principle of locality*. Because the effect of an event is local, there are no explicit global event constraints.

The principle of locality makes it easy to define, for each event, a function on possible worlds which corresponds to it. For example, `inc-age(p0)` defines a function on *PW* which, for each $w \in PW$, replaces the version in *w* identified by `p0` with a version in which `age` is increased by 1.

3.2. Event preconditions

An event is a function on version sorts, and it can happen that it maps an admissible object version to an inadmissible one, for example in the application of `inc-age` to a person version with age 149. And even if it maps an admissible version to an admissible version, it may map an admissible world to an inadmissible world. For example, if `get-spouse(p1, p2)` is applied to a person, it will in general lead to a world where `[GAC1]` is not satisfied. We call an event application which maps an admissible to an inadmissible world *inadmissible*. An event application is inadmissible because, when executed, it would cause a violation to a local or global attribute constraint. In CMSL we can define *event preconditions* to prevent an attribute constraint from being violated. An event precondition is a set of equations attached to an event such that the event cannot be executed in a world if the equations are not satisfied in that world. Like attribute constraints, event preconditions are divided into local and global preconditions. When an attribute constraint violation is prevented by a precondition, we say that the attribute constraint is *preserved* by the precondition, or that the attribute constraint is *invariant* under the event application to which the precondition is attached. The following preconditions preserve the local constraints in `Persons5`.

```
object spec Persons6
    ; all sections as in Persons5
    local event preconditions
[LEP1]  inc-age(p) when age(p) < 149 = true
[LEP2]  get-spouse(p1, p2) when p1 eq p2 = false
```

end spec Persons6

A global event precondition which preserves the global constraint on spouses will be given below, in the section on communication. Preconditions play no role in the expansion of the object specification to the version algebra specification. They will get a semantics when introducing processes below. Here, we can simply assume that each event application that ever occurs satisfies its preconditions.

It then remains to be shown that the preconditions of an event preserve the attribute constraints. This can be proved in the following way for local event preconditions; the proof technique for global event preconditions is similar and is explained briefly after we introduced communication. To prove that a local event precondition preserves a local attribute constraint, we expand the precondition just as we expand local attribute constraints. The resulting equation is then true or false of elements of the version algebra. Suppose that `pc` is a closed term of sort `ADM-PERSON-VERSION` which satisfies the expanded form of `[LEP1]`. So we know that

`age(pc) < 150 = true` (because `pc` is admissible) and
`age(pc) < 149 = true` (because it satisfies `[LEP1]`).

Then we must prove that after application of `inc-age`, `[LEP1]` still holds. So we must prove that

```
age(inc-age(pc)) < 150 = true
```

holds, using the two equations assumed above and the equations in the version algebra specification. The proof is trivial and is not given here. Note that we use the rules of equational deduction and, because we work with an initial algebra, we can also use structural induction. This is one of the advantages of expanding the object specification to an equational value specification and using an initial semantics for the expanded specification.

3.3. Processes

Each object executes a *process* composed of the events declared in its object specification. To specify processes and give a semantics to these specifications, we use the theory of process algebra developed elsewhere [5, 7, 27, 28]. The particular family of process algebras we use is called ACP (Algebra of Communicating Processes) developed by Bergstra & Klop [1, 3]. To use this in a CM specification, we must first select a set of operators which compose events into processes, select an appropriate set of axioms for these operators, and finally decide on an appropriate semantics for them. We call the specification of operators and axioms for them a *process theory* and the intended model of this the *process model* of the theory. Given a process theory and an intended process model, we can then specify the particular process executed by an object by a set of equations over the process theory. This set of equations is usually called a *recursive specification* in process algebra, but to avoid ambiguity in the use of the word "specification," we use the term *process query*. The intention is that the process query has precisely one solution in the process model. For each class of objects, we will specify a process query which defines the process executed by objects of that class.

To specify this in CMSL, we add PSL (process specification language) to VSL and OSL. PSL has virtually the same syntax as VSL, but starts its specifications with the keyword **process spec** instead of **value spec**. As an example, take the following theory of *basic process algebra,* which defines operators `+` and `.` for *choice* and *sequence,* respectively.

process spec BPA
  sorts
```
     EVENT < PROCESS
```
  functions
```
     inc-age         : EVENT
```

```
        change-address : EVENT
        get-spouse     : EVENT
        _ + _  : PROCESS x PROCESS -> PROCESS
        _ . _  : PROCESS x PROCESS -> PROCESS
    variables
        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)
end spec  BPA
```

All process theories in ACP have the sorts of events and of processes. Each event is a process. `BPA` declares the events of `Persons5` as constants, because it is intended for use in the persons specification. In general, `BPA` is parametrized by its events. When `BPA` is used in a CM specification, it should contain all event names declared in the CM specification as constants of sort `EVENT` (this explains the event name uniqueness requirement for object specifications).

The above process theory can be imported in `Persons5`, because all events applicable to persons are declared in it. We can now build terms like

(1) `inc-age . (change-address + get-spouse)`

(2) `inc-age . change-address + inc-age . get-spouse`

(3) `(change-address + get-spouse) + change-address`

(1) describes a process that first does an `inc-age` and then has a choice between doing `change-address` or `get-spouse`. We call this an *arbitrary* choice, because it is not (and cannot) be specified which branch is taken. (2) describes a process which has the choice of doing `inc-age . change-address` or doing `inc-age . get-spouse`. This is called a *non-deterministic* choice, because doing `inc-age` may lead to a state in which `change-address` may be done or to a state in which `get-spouse` may be done. Doing `inc-age` thus leads to one out of a *set* of states. According to `[A1-3]`, (3) offers a choice of `change-address` and `get-spouse`, because we have

```
(change-address + get-spouse) + change-address =
change-address + get-spouse + change-address =
change-address + get-spouse =
get-spouse + change-address.
```

There is no axiom

$$x(y+z) = xy + xz,$$

because then a deterministic process like (1) would be equated with a nondeterministic process like (2). As argued by Milner [27], there are UoD's where the distinction between a deterministic and a nondeterministic process is important, so our process theory should be able to distinguish these processes. For UoD's where the distinction is immaterial, just add the above axiom to the theory. OBLOG [10] uses a trace semantics of processes, [30], in which deterministic and nondeterministic processes are identified.

There is a large variety of models for ACP. We choose the *graph model,* because this is a theoretically well-understood model and because its elements can be represented pictorially in an intuitively understandable way. This is an important asset in conceptual modeling. In the graph model, processes (1) and (2) above would be represented as shown in figure 3.

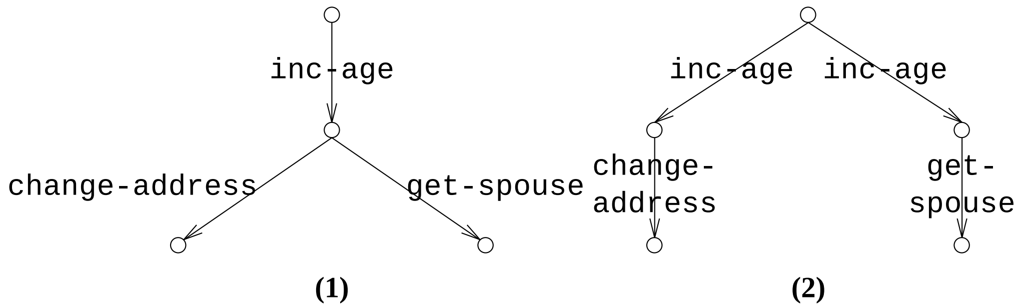

Figure 3.

Figure 3 also illustrates the difference between the two processes, which is that the moment of choice is different in them.

In an earlier paper, [36], we showed that in conceptual models we should distinguish *generic* processes from *individual* processes. A generic process is composed from the names of events declared in the **events** section. The example processes above are therefore generic. An individual process, on the other hand, is composed of *applications* of those events. It thus presupposes a different process theory, where the event names are all possible closed terms $e(c_1, \ldots, c_n)$. For example, the person identified by the closed term `pc` of sort `PERSON` executes an individual process composed of event applications like `inc-age(pc)`, `change-address(pc, ac)` etc., where `pa` is a closed term of sort `ADDRESS`.

An individual process is an *instance* of a generic process, and each particular object executes an individual process. All objects identified by an identifier of sort $s \in I$ execute an instance of the same generic process, which is therefore associated to s. The generic process of s is called `s`. The relation between `s` and the individual processes which are its instances can be syntactically described by the generalized state operator. This operator is introduced by Baeten & Bergstra [2]. In [36] we show how to apply this to the specification of dynamic objects, and [37] a detailed study of the instantiation relation between processes is made. Here, we merely give an example.

The generic process `inc-age . change-address` can be instantiated to the individual process `inc-age(p0) . change-address(p0, a)`, where `p0` is a closed term of sort `PERSON`, and `a` is a variable of sort `ADDRESS`. Figure 4.a shows the generic process and figure 4.b shows the individual process executed by the object identified by `p0`.

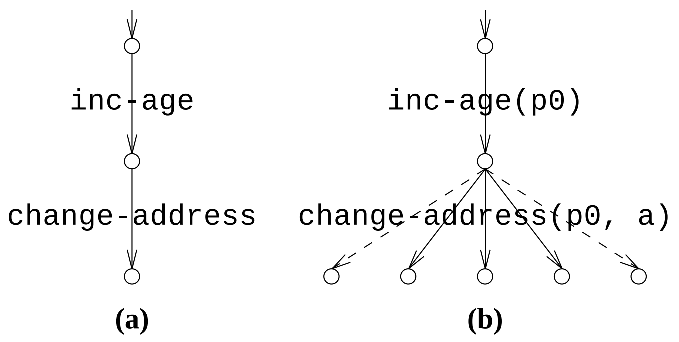

Figure 4.

When `p0` executes `change-address`, it executes one out of an infinite set of possible event applications `change-address(p0, a)` for `a` a closed term of sort `ADDRESS`. Figure 4.b shows one edge for each possible value of `a`. The effect of executing an event on the attribute values of an object is not shown in figure 4.

There are thus two process theories we need, one of generic processes and one of individual processes. Each object specification imports the generic theory and contains a **process** section in which a particular generic process `S` is associated to each identifier sort *s*. We do this by writing down a process query consisting of 1 or more equations, using the identifier sort names as main variable, the event names as constants, and the process operators as functions. In the persons example, we may specify:

```
object spec Persons7
    import
        Addresses2, Naturals, PrimitiveIdentifiers, BPA
    ; The other sections identical to those in Persons6
    process
[P]     PERSON = (inc-age + change-address + get-spouse) . PERSON
end spec Persons7
```

This is a trivial process which iterates over a choice of the available person actions. After each action, it returns to the starting state (see figure 5).

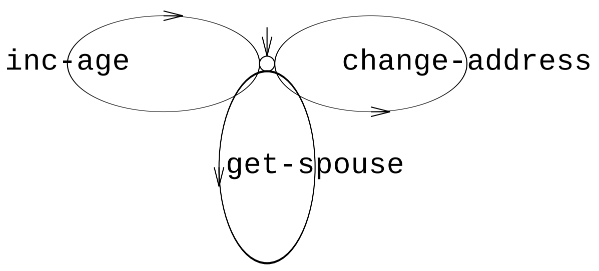

Figure 5.

The process query should have a unique solution in the process model. There are syntactic criteria which allow one to decide whether the process equations have a unique solution, see the references on process algebra for this. Since the constants in the process query are event names, the specified process is generic; individual objects of sort *s* executes an instance of this process. In CMSL we follow the convention that, if no process is explicitly specified for *s*, then `S` is an iteration over all events executable by objects of sort *s*.

We merely mention the problem of formulating consistency requirements on the generic processes specified for identifier sorts s_1 and s_2, if $s_1 \leq s_2$. In [37], we propose the requirement that S_2 be a *process abstraction* of S_1. Roughly, this says that the more general a process is, the smaller the set of events is occurring in it, and the smaller the set of observations is one can make of the process. Abstraction is a relation between processes defined in process algebra. Ehrich et al. [11] propose a similar criterion in the context of a trace semantics. Discussion of the suitability of this criterion goes outside the scope of this paper.

3.4. Parallelism

If we extend BPA with operators for parallel composition of processes, and their axioms, we get an axiom system called ACP (i.e. the same name as the family of axiom systems ACP). Three important operators are $\|$ (parallel composition, also called merge), $\lfloor\!\lfloor$ (left merge), and $|$ (synchronous communication). The process $x \| y$ is the parallel composition of processes x and y, $x \lfloor\!\lfloor y$ is that parallel composition of x and y whose first event comes from x, and $x | y$ is the process which starts with a synchronous execution of an event from x with an event from y. Some typical axioms are

```
[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 . y) = (e1 | e2) . (x || y)
```

These axioms imply a choice for a certain model of parallelism, called *interleaving*. `[CM1]` implies that in a parallel execution of two processes, at each step the next event is an event from x, or from y, or a synchronous occurrence of the first events from x and y. So in the execution of $x \| y$, the event occurrences are totally ordered. This contrasts with a *partial order* model of parallelism, in which event occurrences need not have a total ordering. See [4] for a comprehensive set of tutorials on these issues. In [36] we discuss the suitability of an interleaving model of parallelism for conceptual models. If a particular UoD requires a partial order semantics of parallelism, then the process theory should be adapted and a partial order model should be chosen.

Using parallel composition, one can specify more complex processes. The following example specifies part of a CM which represents a UoD of the readers of a newspaper called The Daily Racket, who can enter submissions to a competition as often as they want, once they have subscribed to the newspaper. A hidden rule of the game is that each reader can win only once [23, 33].

```
object spec DailyRacket
    import
        PrimitiveIdentifiers, ACP
    identifier sorts
        READER specializing PRIM
    functions
        r0 : READER
    events
        win    : READER -> READER
        submit : READER -> READER
    process
[P1]    READER = SUBMIT || win
[P2]    SUBMIT = enter . SUBMIT
end spec DailyRacket
```

This specification imports `ACP`, which is assumed to declare the events `win` and `submit` as constants. These are events in the life of a reader, as declared above, and need no parameters (another hidden rule of the game is that any reader who has not yet won, can win, regardless the quality of the submission). `SUBMIT` is a second variable in the process query. Process variables occurring in the process section need not be declared, because they all range over the sort `PROCESS`.

Local and global communication

The Daily racket example is incomplete, because it should involve communication between different objects. ACP defines *synchronous communication,* which consists of a synchronous execution of 2 or more events. Asynchronous communication can be modeled by adding a mailbox between sender and receiver, with which sender and receiver communicate synchronously. There is no direction in the communication, but this can be added if required [6].

If an object may communicate with itself when it executes two parallel processes. This is called *local communication.* If two or more different objects (i.e. with different identifiers) participate in a communication, then the communication is called *global.*

The specification of communication consists of two parts.

1. Specify *messages.* These are events that can only occur as part of a communication.
2. Specify *communications.* These are events which consist of a synchronous composition of other events.

Note that the component events in a communication may also be able to occur independently, because they are not required to be messages. This allows the specification of a certain asymmetry in communication. We do not go into this here.

There must not be any event constraints for communications. The intention is that the effect of a communication on attribute values is completely defined by the effect of its participating events. This is formalized in the specification expansion.

The following example gathers all elements of the person specification we have seen, and adds a communication event (`marry`). It also declares `get-spouse` to be a **global message**, and defines `marry` to be a synchronous execution of two instances of `get-spouse`.

```
object spec Persons8
    import
        Addresses2, Naturals, PrimitiveIdentifiers, ACP
    identifier sorts
        PERSON specializing PRIM
    functions
        p0  : PERSON
    attributes
        age     : PERSON -> NAT
        address : PERSON -> ADDRESS
        spouse  : PERSON -> PERSON
    events
        inc-age        : PERSON           -> PERSON
        change-address : PERSON x ADDRESS -> PERSON
        get-spouse     : PERSON x PERSON  -> PERSON global message
        marry          : PERSON x PERSON  -> PERSON
    variables
        p, p1, p2 : PERSON
        a : ADDRESS
    global communications
[C]     marry(p1, p2) = get-spouse(p1, p2) | get-spouse(p2, p1)
    local event constraints
[LEC1]  age(inc-age(p)) = age(p) + 1
[LEC2]  address(change-address(p, a)) = a
[LEC3]  spouse(get-spouse(p1, p2)) = p2
```

```
      local attribute constraints
[LAC1]  spouse(p) eq p = false
[LAC2]  age(p) < 150 = true
      global attribute constraints
[GAC1]  spouse(spouse(p)) = p
      local event preconditions
[LEP1]  inc-age(p)
          when age(p) < 149 = true
[LEP2]  get-spouse(p1, p2)
          when p1 eq p2 = false
      process
[P]     PERSON = (inc-age + change-address + get-spouse) . PERSON
end spec Persons8
```

By flagging `get-spouse` as a global message, we indicate that it cannot occur, except as part of an execution of a global communication. `[C]` says that `marry` is such a global communication, and since it is the only communication, `get-spouse` can only occur as part of `marry`. The semantics of global communications is defined in such a way that the subject variables of the messages participating in a global communication must be different. Thus, in any execution of `marry(p1, p2)`, we have `p1 eq p2 = false`.

This takes us back to attribute constraint invariance. We saw earlier that a single execution of `get-spouse` could cause a violation of `[GAC1]`. By the above arrangement, this violation cannot occur. In the specification expansion, the defining equations for `marry` can easily be defined, and these can then be used to prove that `marry` preserves `[GAC1]`.

CMSL also allows **global preconditions**, which are preconditions of global communications. Global communications and their preconditions are the mechanism whereby global attribute constraints must be preserved in CMSL.

4. Other topics

4.1. Roles

CMSL allows the specification of two important structures not yet mentioned, roles and existence. A *role* is a class whose membership is not essential. For example, the class of students is such that its elements could have existed even if they were not students. On the other hand, a person cannot exist without being a person. This is the reason for calling the class of persons a *natural kind;* it expresses something of the essential structure of the UoD [29, 39]. Essential membership of natural kinds is represented in CMSL by the fact object identifiers are declared as having a sort essentially. Contingent membership of a role is represented as follows. Roles are declared in a separate section and must have an identifier sort as supersort, for example

```
      roles
          STUDENT < PERSON
```

We may now declare attributes for roles, such as `stud# : STUDENT -> STUD#`. In the version algebra, there will be a version sort `STUDENT-VERSION` which is a subsort of `PERSON-VERSION`. This means that the set of possible person version is extended with the set of al possible student versions, and that a person event can map a person version to `STUDENT-VERSION`. We say that that person then *plays* role `STUDENT`. Similarly, there may be events which map student versions to person versions not playing the student role.

Note that we should not have any object identifiers of sort `STUDENT`. If there were, then the objects identified by those identifiers would be students essentially, and `STUDENT` would thereby cease to be a role. There are many restrictions on the declaration of roles, which we cannot go into here.

We may define a process query for students. Assuming the abstraction requirement on processes, the process executed by students contains the process executed by persons as an abstraction. This means that we should specify explicitly how the student process is embedded in the person process.

4.2. Existence

A second important topic which should be mentioned is that of existence. In CMSL specifications, there is an object identifier `db` which has an attribute `ext-s` for each identifier sort `s`. Its value is the set of existing identifiers of that sort. The object identified by `db` is called the *existence monitor*, and this executes a process that iterates over creation and deletion events of object identifiers, which are simply additions to and removals from `ext-s` for the appropriate `s`. The CM is initialized with the existence monitor in a certain state, and whenever this executes a creation event, the created object is initialized and starts running its process in parallel to the processes already running. A process creation operator for ACP is defined in [8], and its application to object creation is given in [37]. Using the existence monitor, one can specify existence constraints such as

```
(p in ext-PERSON(db)) implies (address(p) in ext-ADDRESS(db)).
```

`implies` is the Boolean operator one expects it is, specified in `Booleans`.

4.3. Transactions and queries

Finally, having specified a CM which is an abstract representation of a UoD, we may want to query this CM. Since the CM is infinite (for example, it contains the natural numbers), we cannot do this in reality, but we can approximate it by an implementation. This is done, of course, by storing the CMS and an initial world of the CM, and then manipulating the world according to the CMS. We define an *interaction* with the CM implementation as a sequence of transactions and/or query executions. A *transaction* is the execution of a local or global event. Note that it may be a synchronous execution of one or more more elementary events. If these events are all executed by different objects, their synchronous execution may be simulated by executing them in sequence. Events are atomic, and this sequence should be considered as an atomic event, as transactions indeed are in database use.

Transactions bring us from the initial world of the CM to one of the possible worlds. Between transactions, one of the possible worlds is current. A *query* is a set of equations over variables of identifier sorts which should be solved in the current world. A simple example of a query is

```
age(p) < 12 = true
```

with `p` of sort `PERSON`, which requests the set of persons under 12 years of age. A query always yields a set of identifiers. The result can be requested by just asking for the possible values of `p` that satisfy the above equation in the current world, or by asking, say, the value of `name(p)` for the possible values of `p`.

The result of a query can also be viewed as a subset of the identifiers existing in a current world. This subset is itself a world and can be subjected to more queries. Queries in CMSL thus display the important closure property that relational queries also have.

5. Conclusions and comparison with related approaches

CMSL allows the specification of objects with static structures of classification, roles, aggregation, generalization and higher-level objects, and with the dynamic structures of events, preconditions, processes, parallelism and communication. CMSL specifications have a declarative semantics, consisting of a version algebra in which attributes are projection functions and events are functions on object versions that keep the object identifier invariant. We can use equational logic to reason about the version algebra, and by initiality, we can use structural induction. The version algebra contains objects as subalgebras and can be used to define possible worlds. Processes have a semantics in a different model, which we take from process algebra. Events are interpreted as functions in the version algebra, or as functions on possible worlds.

Current work on CMSL includes construction of a workbench with which linguistic and graphical CM specifications can be edited and manipulated, and an animator which brings the specifications to life by simulating a small CM. We intend to do further work on process generalization, for which the abstraction relation does not seem to be the optimal formalization. Other topics for further work include the definition of a modal equational logic, with events as modal operators, which can do the same for us in object specifications as equational logic does in expanded specifications, and the definition of application models, which are particular views of the CM, geared to a particular application domain.

Acknowledgements: Reind van de Riet provided the occasion to do the work which led to this paper and participated in many discussions on dynamic conceptual models. Frank Dignum, Wiebren de Jonge, Jan Willem Klop, John-Jules Meyer and Hans Weigand all contributed to the ideas expressed in this paper. Hans Weigand eliminated some unclarities from a preliminary version.

6. References

1. Baeten, J.C.M., *Procesalgebra,* Kluwer (1986).
2. Baeten, J.C.M. and Bergstra, J.A., ‘‘Global Renaming Operators in Concrete Process Algebra,’’ *Information and Control*, pp. 205-245 (1988).
3. Baeten, J.C.M., *Process algebra,* North-Holland (to appear).
4. Bakker, J.W. de, Roever, W.-P. de, and Rozenberg (eds.), G., *Linear Time, Branching Time and Partial Order Logics and Models for Concurrency*, Springer (1989). Lecture Notes in Computer Science 354.
5. Bergstra, J.A. and Klop, J.W., ‘‘Algebra of Communicating Processes with Abstraction,’’ *Theoretical Computer Science* **37**, pp. 77-121 (1985).
6. Bergstra, J.A., ‘‘Put and Get, Primitives for Synchronous Unreliable Message Passing,’’ Report LGPS 3, University of Utrecht, Department of Philosophy (1985).
7. Bergstra, J.A. and Klop, J.W., ‘‘Algebra of Communicating Processes,’’ pp. 89-138 in *Mathematics and Computer Science (CWI Monographs 1)*, ed. J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, North-Holland (1986).
8. Bergstra, J.A., ‘‘A Process Creation Mechanism in Process Algebra,’’ pp. 81-88 in *Applications of Process Algebra (CWI Monograph 8)*, ed. J.C.M. Baeten, North-Holland (1989).
9. Bergstra, J.A., Heering, J., and Klint, P., *Algebraic Specification,* ACM Press/Addison Wesley (1989).
10. Costa, J.F., Sernadas, A., and Sernadas, C., *OBL-89 User’s Manual, version 2.3,* Instituto Superior Técnico, Lisbon (may, 1989).
11. Ehrich, H.-D., Sernadas, A., and Sernadas, C., ‘‘Objects, Object Types, and Object Identification,’’ pp. 142-156 in *Categorical Methods in Computer Science*, ed. H. Ehrig, H. Herrlich,

H.-J. Kreowski, and G. Preuß, Springer (1987). Lecture Notes in Computer Science 393.

12. Ehrich, H.-D., Sernadas, A., and Sernadas, C., *From Data Types to Object Types*, submitted. 1989.

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

14. Fiadeiro, J. and Maibaum, T., ''Temporal Reasoning over Deontic Specifications,'' Technical Report, Department of Computing, Imperial College (1989).

15. Fiadeiro, J., Sernadas, C., Maibaum, T., and Saake, G., *Proof-Theoretic Semantics of Object-Oriented Specification Constructs*, To be published, IFIP TC2 Working Conference on Database Semantics, Windermere, U.K. (2-6 july 1990).

16. Goguen, J.A., Jouannaud, J.-P., and Meseguer, J., ''Operational Semantics for Order-Sorted Algebra,'' pp. 221-231 in *12th International Coloquium on Automata, Languages and Programming*, ed. W. Brauer, Springer Lecture Notes in Computer Science 194 (1985).

17. Goguen, J.A. and Meseguer, J., ''EQLOG: Equality, Types, and Generic Modules for Logic Programming,'' pp. 295-363 in *Logic Programming: Functions, Relations, and Equations*, ed. D. DeGroot & G. Lindstrom, Prentice-Hall (1986).

18. Goguen, J.A. and Meseguer, J., ''Order-Sorted Algebra Solves the Constructor-Selector, Multiple Representation and Coercion Problems,'' *Second IEEE Symposium on Logic in Computer Science,* Ithaca, New York, pp. 18-29 (june 1987).

19. Goguen, J.A. and Meseguer, J., ''Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics,'' pp. 417-477 in *Research Directions in Object-Oriented Programming*, ed. B. Shriver & P. Wegner, MIT Press (1987).

20. Goguen, J.A. and Winkler, T., ''Introducing OBJ3,'' SRI-CSL-88-9, Stanford Research Institute Inc. (August 1988).

21. Goguen, J.A. and Meseguer, J., *Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations,* Programming Research Group, Oxford, and SRI International, Menlo Park (June 19, 1989).

22. Goguen, J.A. and Wolfram, D., *On Types and FOOPS*, To be published, IFIP TC2 Working Conference on Database Semantics, Windermere, U.K. (2-6 july 1990).

23. Jackson, M., *System Development,* Prentice-Hall (1983).

24. Maier, D., Rozenshtein, D., and Warren, D.S., ''Windows on the World,'' *Proceedings, ACM/SIGMOD International Symposium on the Management of Data*, San Jose, California, pp. 68-78 (1983). Sigmod Record, Vol. 13, no.4.

25. Maier, D., Ullman, J.D., and Vardi, M.Y., ''On the Foundations of the Universal Relation Model,'' *Transactions on Database Systems* **9**, pp. 283-308 (june 1984).

26. Meyer, J.-J. Ch., Weigand, H., and Wieringa, R.J., ''A Specification Language for Static, Dynamic and Deontic Integrity Constraints,'' pp. 347-366 in *2nd Symposium on Mathematical Fundamentals of Database Systems*, ed. J. Demetrovics, B. Thalheim, Springer, Visegrád, Hungary (june 1989). Lecture Notes in Computer Science 364.

27. Milner, R., *A Calculus of Communicating Systems*, Springer (1980). Lecture Notes in Computer Science 92.

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

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

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

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

32. Ullman, J. D., ‘‘The UR Strikes Back,’’ *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, Los Angeles, pp. 10-22 (march 29-31, 1982).

33. Wieringa, R.J., ‘‘Jackson System Development Analysed in Process Algebra,’’ IR-148, Department of Mathematics and Computer Science, Vrije Universiteit (march 1988).

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

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

36. Wieringa, R.J. and Riet, R.P. Van De, ‘‘Algebraic Specification of Object Dynamics in Knowledge Base Domains,’’ pp. 411-436 in *Artificial Intelligence in Databases and Information Systems (DS-3)*, ed. R.A. Meersman, Zhongshi Shi, Chen-Ho Kung, North-Holland (1990).

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

38. Wieringa, R.J., *An Algebraic Semantics for Dynamic Objects. Extended abstract*, Submitted. March 1990.

39. Wiggins, D., *Sameness and Substance,* Basil Blackwell (1980).