Integrated Specification of Values, Objects and Processes for Object-Oriented Models

EXTENDED ABSTRACT

Roel Wieringa
Department of Mathematics and Computer Science
Free University
De Boelelaan 1081
1081 HV Amsterdam

uucp: roelw@cs.vu.nl

Contents
1. Introduction
   1.1 The goal of the research
   1.2 The results so far
2. The structure of an object-oriented CM
   2.1 Object identifiers
   2.2 Object states
      2.2.1 Attributes
      2.2.2 Roles
      2.2.3 Processes
3. Updates and equational queries
4. Current and future work

1. Introduction

1.1. The goal of the research
The goal of the research summarized in this abstract is first, to formalize object-oriented models and second, to develop a method for producing these models. This goal can be stated in semi-formal form as the solution of the following equations for the unknown terms that appear on the right-hand side of an equation, but do not appear at a left-hand side.

(1) method = specification language + semantics + pragmatics
(2) pragmatics = tasks + task ordering + heuristics

By ‘‘method’’ we mean a method to develop a conceptual model (CM) of a universe of discourse (UoD). The specification language should be formal, and each consistent specification has an intended denotational model which can serve as a CM of a UoD. The semantics should include a significant portion of the structures that are found in the "semantic data models” studied in the past and in object-oriented models currently being developed. The pragmatics give advice on how to go about finding the relevant structures in the UoD and specify a model of them in the specification language.
This is basically a set of tasks to be executed by the analyst/designer. Often, there should also be an advice on the ordering of tasks as well as a set of heuristics that help the analyst make decisions during the modeling process.

1.2. The results so far
So far, we have developed an equational specification language, called CMSL, for object-oriented CM’s. As a bonus, a minor extension gave us a data-manipulation and a query language for object-oriented databases, although this was not part of our original project goal. Finally, we have a syntax-directed editor for the CMSL. We briefly discuss each of these results, and give a list of current and future research afterwards.

2. The structure of an object-oriented CM
The CM’s specifiable by CMSL have the following characteristics [27]. Each CM consists of a set of possible objects, and each object consists of a unique object identifier and a state space. The elements of a state space are called states. There is no way to indicate the current state of a single object. Instead, we define a world as a function from object identifiers to state spaces such that each identifier is mapped to an element in its own state space. Thus, in one world, all objects are in a state. They may even be in the same state, but they all have different identifiers. By default, each CMSL specification declares an identifier db whose state contains, among others a set of identifiers called ext(db). By definition, the identifiers in this set identify existing objects. ext(db) is called the existence set.

2.1. Object identifiers
One part of CMSL is called VSL (Value Specification Language) and can be used to specify abstract data types. VSL is an extension of ASF, a many-sorted equational specification language [6], with two features from OBJ viz. the capability to specify a partial ordering on sorts and the capability to specify sort constraints [15]. VSL is given the standard initial algebra semantics described in the cited papers.

Values of any type can be used as object identifiers. Suppose that we specify the type PERSON in VSL by

```
value spec PersonIds
  import
    Booleans
  sorts
    PERSON
  functions
    p0 : PERSON
    next : PERSON -> PERSON
    eq : PERSON x PERSON -> BOOL
  variables
    p, p1, p2 : PERSON
  equations
    [E1]   eq(p0, p0) = true
    [E2]   eq(next(p1), next(p2)) = eq(p1, p2)
    [E3]   eq(p0, next(p)) = false
    [E4]   eq(next(p), p0) = false
end spec  PersonIds
```
Then we declare \texttt{PERSON} to be an identifier sort by

\begin{verbatim}
object spec PersonObjects
  import   
      PersonIds

  identifier sorts
      PERSON

end spec PersonObjects
\end{verbatim}

Any set of sorts that is downward closed with respect to the partial ordering on sorts can be declared to be identifier sorts. Object specifications are given in \textit{OSL}, which is another part of \textit{CMSL}, next to \textit{VSL}. Object specifications are given a semantics by expanding them textually to value specifications, which then have the standard initial algebra semantics. In the example, the expansion will cause a sort \texttt{PERSON-VERSION} to be declared, whose elements have the form \( \langle p, \langle \rangle \rangle \) for each data element \( p \) of type \texttt{PERSON}. This means that each \( p \) identifies a person with the trivial internal state indicated by \( \langle \rangle \). Thus, in this example the sort of person versions is isomorphic to the sort of person identifiers.

2.2. Object states

In general, any value sort not explicitly declared as identifier sort can be viewed as identifying trivial objects like person objects above. There are three ways to give an object a non-trivial internal state, by declaring attributes for its identifier sort, by declaring roles that it can play, and by declaring a process for it. We discuss each of these in turn.

2.2.1. Attributes

An attribute is a function from an identifier type to any value type. For example,

\begin{verbatim}
object spec PersonAttributes
  import   
      PersonObjects, Addresses, Naturals, Strings

  attributes
      name : PERSON -> STRING
      address : PERSON -> ADDRESS
      age : PERSON -> NAT

  variables
      p : PERSON

  [LAC1] age(p) < 150 = true

end spec PersonAttributes
\end{verbatim}

We presuppose an object specification \texttt{Addresses} and two value specifications \texttt{Naturals} and \texttt{Strings}. The result type of an attribute may itself be an identifier sort (e.g. \texttt{ADDRESS}), so that an \textit{object aggregation graph} is defined. This graph may contain cycles, such as if we had defined the attribute \texttt{manager : PERSON -> PERSON}. We can also have set-valued attributes or sets as object identifiers. Suppose we have a specification \texttt{Sets} parametrized by \texttt{Items}, which is instantiated with persons as items. Let \texttt{PERSONS} be the type of finite sets of persons, then we can declare this to be an identifier sort and declare the attributes

\begin{verbatim}
  children : PERSON -> PERSONS
  avg-age : PERSONS -> RAT
\end{verbatim}
The value of \( \text{avg-age}(\{p_1, \ldots, p_n\}) \) is a rational number which should be tied via a global constraint to the bag of values \( \text{age}(p_1), \ldots, \text{age}(p_n) \).

A local attribute constraint limits the set of admissible attribute values for single objects. Thus, \( \langle p, \langle \text{name: } n, \text{address: } a, \text{age: } 160 \rangle \rangle \) is an inadmissible person version. The formal semantics of the example object specification is that it is expanded to a value specification containing, among others, the sorts \( \text{ADM-PERSON-VERSION} \) and \( \text{PERSON-VERSION} \), such that the first of these is a subsort of the second. The local attribute constraints are translated into sort constraints for \( \text{ADM-PERSON-VERSION} \), so that this sort contains precisely the person versions that satisfy the local constraints.

One may also define global attribute constraints, which limit the admissible combinations of attribute values of sets of objects. An example of a global constraint is \( \text{age(manager(p))} > \text{age(p)} \) = true. This limits the set of admissible ages of a person and his or her manager. In general, this is a global constraint.

Each possible world can be thought of as a set of object versions \( \langle p, \sigma \rangle \) for identifiers \( p \) and attribute vectors \( \sigma \), such that each \( p \) occurs precisely once in a world. An admissible world can then be defined as a possible world that satisfies the local and global attribute constraints.

### 2.2.2. Roles

Identifier sorts give a classification of objects according to their essential characteristics. If \( p \) is an identifier of sort \( s \), then it has this sort in all possible worlds, so it has the attributes defined for \( s \) in all possible worlds, and is subject to the constraints declared for \( s \) in all possible worlds. The object identified by \( p \) could not exist without being classified as an \( s \) or without having the structure defined for objects of this class.

We also need classes that classify objects contingently. For example, the class of students consist of those persons which happen to be students, but the objects classified as students can very well exist without being a student. If persons are students, they have the attributes and are subject to the constraints defined for students, but there are worlds where they are not students. This contrasts with the class of persons. An object classified as a person cannot exist without being a person. We call contingent classes roles, and an object in one of these classes is said to play a role. Next to the tuple of attribute values of an object, the roles that an object is playing are also part of the state of an object. Roles in the sense we have defined them here were already discussed (albeit in the context of a network DB schema) by Bachman and Daya [2] in 1977. An example of a role specification in CMSL is

```cmsl
object spec PersonRoles
  import
    PersonAttributes
  roles
    SPOUSE < PERSON
    STUDENT < PERSON
  attributes
    spouse : SPOUSE -> SPOUSE
  variables
    p : spouse
  global attribute constraints
    [GAC1] spouse(spouse(p)) = p
end spec PersonRoles
```

To give this a formal semantics, we expand it to a value specification containing sort declarations
ADM-SPOUSE-VERSION < SPOUSE-VERSION and ADM-STUDENT-VERSION < STUDENT-VERSION. The elements of SPOUSE-VERSION have the form \((p, \langle \text{name: n, address: a, age: a, spouse: s} \rangle)\). So in this example, student versions are indistinguishable from person versions, because no extra attributes have been defined for students. The admissible version sorts of roles are again subsorts defined by the local attribute constraints. Global attribute constraints are interpreted in the set of possible worlds, where they contribute to defining the set of admissible worlds.

Note that spouses and students take their identifiers from persons. It is a general principle for roles that they have no identifiers themselves but should be declared as a subsort of an identifier sort. This way, we can define the set of possible worlds in such a way that there are worlds in which a person of type PERSON plays no role defined for persons.

Note also that spouses and students have no common subtype. If we want to admit married students, we should explicitly declare a role like SPOUSE-STUDENT as a subtype of spouses and students.

2.2.3. Processes

Each object has a life which is a process composed of elementary events. The third and final part of the internal state of an object is its position in this process. To be able to specify the process an object goes through, we must first specify the elementary events applicable to an object, and the operators by which these events can be composed into processes. An example of a CMSL event specification is

```plaintext
object spec PersonEvents
  import PersonRoles
  events
    inc-age : PERSON -> PERSON
    change-address : PERSON x ADDRESS -> PERSON
    get-spouse : PERSON x PERSON -> SPOUSE

  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 event preconditions
    [LEP1] inc-age(p) when age(p) < 150 = true
end spec PersonEvents
```

Each event is a function of one or more arguments, of which the first is the type of the identifier of the object executing the event. If this sort and the result sort are both identifier sorts, then they must be equal, but if at least one of them is a role, then they must be compatible to each other through the ordering on identifier sorts and roles.

The effect of the event on the attribute values of the object executing it is defined by local event constraints plus a local frame assumption which says that attributes about which nothing is said, are not affected by the event. To give an event specification a formal semantics we expand the object specification to a value specification in which events are functions on version sorts that keep the identifier invariant. The effect of the events on the attribute values is determined by the local event constraints and the local frame assumption. For example, inc-age increases the age field in an object version and keeps all other attribute values invariant.

There are no global event constraints in CMSL. The effect of any event is strictly local (with a
proviso for global communication events, defined below).

For each event we may define local preconditions. The formal meaning of these is that an event simply cannot occur if these conditions are not satisfied by the object executing the event. Any attempt to execute an event that does not satisfy the local preconditions, deadlocks. Local preconditions can be used to prove, in the value specification which is the expansion of the object specification, that the local attribute constraints remain invariant under event applications that satisfy the local preconditions.

The global constraint $[GAC1]$ in PersonRoles can be violated by get-spouse, which may cause a person $p$ to be married to someone whose spouse is not $p$. The solution to this is to block get-spouse($p1$, $p2$) in all cases except when it occurs synchronously with a corresponding get-spouse($p2$, $p1$). More on this in a bit. A synchronous execution of events by different objects is called a global communication, and may itself be subject to global preconditions. Global preconditions can be used to prove that global constraints remain invariant under global communication events.

Next to the elementary events, we need to specify the operators by which we can compose events into processes. For this we use a third part of CMSL, called PSL (Process specification language), which is based on the algebra of communicating processes developed by Bergstra & Klop [5]. For example, the operators $+$ (choice) and $\cdot$ (sequential composition) are defined as follows.

```plaintext
process spec BasicProcessAlgebra
parameters
    Events begin
sorts
    EVENT
end Events
sorts
    EVENT < PROCESS
functions
    _ + _ : 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 BasicProcessAlgebra
```

This parametrized specification should be instantiated for each identifier sort and role, with the events declared for that sort or role as constants of type EVENT. This need not be done explicitly in CMSL.

Other operators that are defined this way are $\parallel$ for parallel composition of processes and $|$ for synchronous execution of the first event of the processes to the left and right of $|$. There are quite a number of models of process algebra, of which we assume the standard graph model. The choice of process axioms and models is up to the information analyst, but for each choice a number of issues must be resolved. One of these is the connection between the process model and the set of possible worlds defined earlier. Roughly, the connection is that the set of possible worlds is turned
into a directed graph, in which the worlds are the nodes and the edges are labeled by event applications to objects. Each event application may change the attribute value or roles of one or more objects in the world to which it is applied, and advances the process (life cycle) of the object one step.

Having specified elementary events and process operators, we can specify the process an object goes through in a set of equations which we call a **process query**. We chose to use the term ‘query’ because a process **specification** is a set of axioms for process operators, like **BasicProcessAlgebra** above. Such a specification has an intended model, from which we now want to select a particular process. A process query is a set of equations to be solved against the intended process model, and which we require to have exactly one solution. There are syntactic criteria for a process query to have precisely one solution in the graph model, given in the work by Bergstra & Klop [5]. An example process query in **CMSL** is

```plaintext
object spec PersonProcess
import
   PersonEvents, ProcessAlgebra
process
[P1] PERSON = (inc-age . PERSON) ||
     ( change-address . PERSON) || get-spouse
[P2] SPOUSE = PERSON || divorce
end spec PersonProcess
```

Variables in the process query are written in capital letters. One of them should be the name of the identifier sort or role for which the process is defined. Objects of that class then execute instances of that process. A single object can execute several processes in parallel, and these processes can communicate through local communications. **CMSL** semantics is built upon the assumption that the events executed by a single object cannot occur synchronously, except when specified explicitly by a local communication. Local communications are dangerous, for through them we can specify mutually inconsistent updates to attribute values.

`get-spouse` is an event in the life of a person, which causes a person to play the role of a spouse. The spouse process is not specified as part of the person process. Instead, the spouse process includes the person process and shows one extra event, `divorce`. This has not been specified, but has a predictable effect. In general, the process for a subclass must include the process(es) declared for its superclasses.

To ensure that `get-spouse(p1, p2)` communicates with `get-spouse(p2, p1)`, we must block its occurrence, except when either of them occurs synchronously with the other. We block isolated event occurrences by declaring them to be messages. For this, the declaration of `get-spouse` in `PersonEvents` must be changed into

```plaintext
get-spouse : PERSON x PERSON -> SPOUSE global message.
```

To specify the context in which `get-spouse` can be executed, we declare a global communication in the conceptual model specification that imports `PersonProcess`.

```plaintext
conceptual model spec PersonModel
import
   PersonProcess
events
   marry : PERSON x PERSON -> SPOUSE x SPOUSE
variables
```
p1, p2 : PERSON

**Global communications**

\[ [GC1] \quad \text{marry}(p1, p2) = \text{get-spouse}(p1, p2) \mid \text{get-spouse}(p2, p1) \]

**end spec** PersonModel

In conceptual model specifications, events may be declared with Cartesian products as output sorts. These are the types of the objects participating in the communication. The semantics of CMSL contains a condition that for global communications, the identifiers of the objects participating in the global communication must all be different.

Global attribute constraints and global event preconditions can also be specified in the CM specification, and in the **initializations** section a set of initial object versions must be listed, which define the initial world of the CM.

3. **Updates and equational queries**

Note that all events, including communication events, are atomic in the sense that they do not lead to intermediate states of the CM. They are, in effect, functions on the set of possible worlds. (Taking preconditions into account, they are functions on the set of admissible worlds.) This means that they can be used as transactions in a DB manipulation language. Furthermore, because global communications update different objects, they may be mapped to a sequence of elementary updates to these objects in the DB, which must be treated as an atomic transaction. To save space, we do not go into this here.

The class of possible worlds can be proven to be a class of possible algebras of attribute specifications, and a DB state represents a class of such algebras by representing only the finite set of existing identifiers and the values of attributes on these identifiers. A query is simply a set of equations to be solved in one of these algebras, followed by one or more expressions to be evaluated with these solutions. For example, \( \text{age}(p) < 15 = \text{true} \) can be solved in an algebra, after which we can ask what the values are of the terms \( \text{name}(p) \), \( \text{age}(p) \) when \( p \) ranges over the solution set. This gives us a powerful way to incrementally specify queries of a DB.

4. **Current and future work**

The work reported in this summary started from a study of the semantic modeling structures found in TAXIS [7], SDM [18] and Galileo [1], and of the object-oriented structures found in Orion [4] and \( O_2 \) [3]. It is heavily based on the concept of object identifier [20]. The algebraic approach to object specification is partly inspired by work done on FOOPS [16] and OBJ3 Some differences with these last approaches are that VSL is more elementary than the value specification language underlying FOOPS and that OSL is centered more explicitly around the concept of an object identifier. Roles and processes are absent from the published versions of FOOPS. Other closely related work is done in the ISCORE project, where the language OBLOG is defined [8, 25] as a test-bed for ideas on the formal specification of objects. In the same project, work on object specification semantics is done from a categorical perspective [10, 11] and on the development of a logic to reason about temporal and deontic properties of objects [12, 14]. This research covers a much wider area than the work done on CMSL. OBLOG has roughly the same capabilities as CMSL as far as the specification of static and dynamic object properties is concerned. Currently, it has no facility to specify roles. On the other hand, the view of attributes in the ISCORE project is more general, for these are defined as functions from event traces to values, and the logical framework to prove temporal and deontic properties of objects is more developed than it is in the case of CMSL.

Our research goal is filling in the unknowns in the equations (1) and (2) given in section 1.1. What we have is a specification language and a semantics which allows one to specify CM’s formally and prove properties (like constraint invariance) of a CM. Current research on CMSL continues...
work towards this goal, and focuses on applications as well as theory.

One kind of application is very practical and consists in the construction of a workbench for CMSL, which should include, in the future, a facility to generate databases. The current status of this part of the project is simply a syntax-directed editor for CMSL. Future work on this workbench will include a prototype object base in which the transactions and queries mentioned in section 3 above can be executed.

The other kind of application consists of testing CMSL on examples. This includes taking examples from diverse system analysis and development methods, like JSD [19], ISAC [22], NIAM [21, 24], and Structured Analysis [9], and translating them into CMSL. This gives us a clue about the kinds of tasks and heuristics mentioned in equation (2) and will bring us closer to achieving our aim of finding a solution to these equations.

Finally, on the theoretical side we plan to start work on an operational semantics for CMSL, and on the definition of a modal logic in which events are modal operators, and which is equivalent to the translation of an object specification into equational logic which we now use. Related work in this area is done by Fiadeiro, Golshani, and Maibaum [13, 17, 14], but we will use a slightly different approach which starts from dynamic logic [23, 26]. We hope this will allow us to add the capability to specify deontic constraints to CMSL.

**Acknowledgements:** This research was partly conducted under the supervision of Prof. R.P. van de Riet. Paul Spruit and Cees Duivenvoorde wrote the syntax-directed editor for CMSL, and discovered some mistakes in the syntax as well as in some example specifications of CMSL.

5. **References**


