A formalization of objects using equational dynamic logic

R.J. Wieringa

Department of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV, Amsterdam
The Netherlands
Email: roelw@cs.vu.nl
Fax: +31 20 427705

Abstract

Order-sorted equational logic is extended with dynamic logic to a specification language for dynamic objects. Special attention is paid to different concepts of encapsulation that play a role in object-orientation. It is argued that the resulting language, CMSL, meets those requirements of the object-oriented database system manifesto [6] that are applicable to object-oriented conceptual models (as opposed to OO databases).

Areas: Integrating logic and object paradigm, formalization of object-oriented concepts

1 Introduction

Object-oriented (OO) database research has started in a bottom-up fashion with implementing a number of working systems, without worrying much about underlying concepts and formalizations. Recently, a top-down movement has started in which, on the one hand, researchers try to formulate unifying concepts that define OO databases and in which, on the other hand, formalizations of OO databases are proposed. Bancilhon [8] and Atkinson et al. in the Object-Oriented Database System Manifesto [6] list a number of features which they argue are essential for OO database systems. Parallel to this, a number of formalizations of OO models have been proposed, such as COL [1, 2, 3], F-logic [33], HILOG [13], ILOG [29], IQL [4, 5], formalizations of $O_2$ [36, 35], OBJ and related languages [20, 21, 22, 23, 25, 38], OBLOG [15, 19, 17, 43], and CMSL [50, 46]. Beeri [9] gives a survey of some issues. Even though these papers are top-down, there is not yet any root concept from which they develop their formalization of object-orientation. The trees of possibilities they explore have different roots and are rapidly evolving into a forest as diverse as the implementations of OO databases that exist now. One important concept that has arisen from the forest of formalization efforts is the concept of an object identifier [4, 17, 29, 31, 34, 45], itself stemming from the concept of a surrogate [14, 27]. In this paper, I try to develop a theory of objects, using the concept of object identifier (oid) as root from which to develop and formalize other concepts. The language defined in this process is called CMSL (Conceptual Model Specification Language) and provides a logic for specifying complex dynamic objects with oid's, and reasoning about them. In contrast to formalization
efforts like IQL, ILOG, HILOG, and F-Logic, it is based on equational logic as used in abstract data type specification. As a result of that, the specification of sets and Cartesian products and of complex structures built from these, is quite natural. Moreover, because equality plays a central role in it, equational logic is excellently suited to reason about object identity. CMSL borrows elements from the equational specification languages ASF [10] and OBJ [23], but differs from them in the central role of object identifiers and in the use of a possible worlds semantics to model object dynamics. CMSL is related in spirit to OBLOG [15], but differs from it in that it does not use a strictly categorical semantics, but uses a more concrete algebraic semantics instead, which in my opinion is simpler, or at least more concrete. Like earlier work done by Maibaum and others [26, 32], CMSL incorporates dynamic logic in equational specification, and builds on work done in algebraic specification.

Older versions of CMSL are described in [45, 46]. The differences with the older version are the use of dynamic logic to specify state changes, which enabled a drastic decrease in the number of different syntactic structures needed to specify dynamic objects, the provision of a sound and complete inference system, and the use of a Kripke structure and an independent process algebra as semantics. This inference system and semantics were first introduced in [41, 48]. In those papers, they were used to for the specification of normative systems. Here, I concentrate on the concepts of object identity and encapsulation, and on synchronous message-passing as a disciplined mechanism to break through encapsulated state and behavior.

The goal of this paper is to show that a number of important features listed in the Manifesto [6, 8] can be specified in CMSL. The aim is conceptual analysis of the concepts used in object-orientation by defining these concepts formally, all the while taking care that the formal definitions have a plausible intuitive interpretation. The aim is not the provision of an executable language or of computation models for this language. However, given the formalization of object-oriented concepts in CMSL, one can use this formalization to prove properties of specifications. For example, the consequences of integrity constraints can be analyzed to see if this is what the user wants, or omissions or inconsistencies in the specification could be discovered. The reason for aiming at conceptual analysis is that we first should be certain that we can specify what we want, before implementing the language or providing operational semantics.

To motivate the object-oriented features chosen from the Manifesto, first note that I restrict myself to OO models, and ignore database issues. A model in this context is a conceptual model (CM) of a piece of reality that is called the universe of discourse (UoD). The model is at the same time a model for a database. By this decision, about half of the mandatory features listed in the Manifesto can be eliminated, because they concern databases rather than models. These are late binding, persistence, secondary storage management, concurrent users, and error recovery.

Having made this restriction, the second remark is that in any approach to object-orientation we can distinguish the specification of the model from the model itself. I regard objects as semantic structures in the model, not as syntactic structures in the specification. An object p is an abstract entity in a model, that represents a real-world entity e in the UoD (figure 1). The remainder of the list of features in the Manifesto can be classified as being applicable to objects (they describe semantic structures) or to the specification of objects (they describe syntactic features), or possibly both. A summary of the semantic structures listed in the Manifesto is that objects

- have a unique identifier,
- encapsulate a local state and behavior,
- can be classified, and
Figure 1: Abstract objects and real-world objects.

- that object complex objects can be built from simpler objects.
- Moreover, classes usually are part of a taxonomic structure.

A summary of syntactic structures is that the specification language must be able to

- allow the specification of new types and of
- ad-hoc queries,
- allow overloading of operation symbols, and be
- computationally complete.

I use this shopping list as a framework for discussion in the rest of the paper.

Section 2 rehearses some of the equational theory of abstract data type (ADT) specification and looks at the extent in which data specified this way meet the demands of our shopping list. It will turn out that the only demands not satisfied are the presence of object identifiers and the encapsulation of local state and object dynamics. Section 3 then shows how to extend this to the specification of dynamic objects in such a way that the demands of our shopping list are met. Section 4 discusses some uses of CMSL, such as the ability to reason about object updates, and some extensions to CMSL that have been realized or are planned for the future. The central role of oid's in this formalization of objects is also discussed. Section 5 concludes the paper.

2 Specifying Values

We assume the theory of order-sorted equational ADT specification developed, among others, by Goguen and Meseguer [24, 25, 44]. This theory is now reaching textbook status [16]. I summarize it very briefly, in order to see how it meets our shopping list.

In order-sorted equational specification, an ADT is a class of models of an equational specification, and an equational specification is a set of axioms in order-sorted first-order logic with equality, using only function symbols and positive conditional equations. As an example, take the following specification of the natural numbers and positive integers, together with some operations.

\[
\text{value spec NaturalNumbers}
\]
sorts
  ZERO \leq NAT
  POSINT \leq NAT

functions
  0 : ZERO
  succ : NAT \rightarrow POSINT
  pred : POSINT \rightarrow NAT
  + - : NAT \times NAT \rightarrow NAT

variables
  n, m : NAT

equations
  \text{E1} \quad \text{succ}(\text{pred}(n)) = n
  \text{E2} \quad \text{pred}(\text{succ}(n)) = n
  \text{E3} \quad 0 + n = n
  \text{E4} \quad \text{succ}(n) + m = \text{succ}(n + m)

end spec \text{NaturalNumbers}

\text{ZERO}, \text{NAT} \text{and} \text{POSINT} \text{are sort names that are interpreted as sets in models of the IntegerValues specification. These sets are called sorts and are the domains of the specified types.} \leq \text{is a partial ordering on sort names that is interpreted as the subset relation on sorts in models of NaturalNumbers. The operations applicable to elements of the sorts are declared in the functions section, and the equations section gives a set of axioms that the operations must satisfy. 0 is a function with no arguments, i.e. it is a constant. Operations defined for a sort are inherited by all its subsorts.}

\text{Definition 1} \text{An order-sorted ADT signature} \text{Sig} = ((S, \leq), F) \text{consists of a partially ordered set} (S, \leq) \text{of sort names, and a set} F \text{of function declarations of the form} f : s_1 \times \cdots \times s_n \rightarrow s_0, \text{with} s_i \in S \text{for} i = 0, \ldots, n. s_1 \times \cdots \times s_n \text{is called the domain of} f \text{and} s_0 \text{the codomain.} n \text{is called the arity of} f.

\text{Terms over} \text{Sig are defined as usual. An equation over} \text{Sig is an unordered pair of two terms over} \text{Sig such that their least sorts are in the same connected component of} (S, \leq). \text{(A connected component of} (S, \leq) \text{is an equivalence class of the symmetric closure of the partial ordering} \leq). \text{An order-sorted ADT specification} \text{Spec} = ((S, \leq), F, E) \text{is an order-sorted signature} \text{Sig} = ((S, \leq), F) \text{and a set} E \text{of equations over} \text{Sig.}

\text{See Goguen and Meseguer [25] for more details. In the sequel, we assume that in all specifications we encounter, every term always has a unique least sort. This is not the case in general, but Goguen and Meseguer show that in regular signatures, every term has a unique least sort. We don't bother to quote the definition of regular signatures here because it does not play a role in this paper.}

\text{An ADT specification is a theory presentation in a subset of order-sorted first-order logic, and can be given an interpretation in a structure as usual. A structure in which the equations of the specification are true is called a model, and a specification is called consistent if it has a model. There are many possible models of a consistent equational specification. A model}\ \text{I of} \text{Spec is called initial if for any model} \text{M of} \text{Spec there is a unique homomorphism from} \text{I to} \text{M. All initial models are isomorphic. The intended semantics of an equational specification is initial. This means that the ADT specified by a specification is the isomorphy class of its initial models.}

\text{One initial model is called the quotient term model and is analogous to the Herbrand model of a logic program. Roughly, it consists of all possible closed terms of the specification, modulo}
provable equality. For example, in the quotient term model, \( \text{ZERO} \) is interpreted as the sort consisting of one element, \([\text{ZERO}] = \{[0]\}\), where \([0]\) is the set (called congruence class) of closed terms provably equal to 0:

\[
[0] = \{0, \text{pred}(\text{succ}(0)), \ldots, 0 + 0, \ldots\}.
\]

Similarly, \( \text{NAT} \) is interpreted as the sort

\[
\{[0], [\text{succ}(0)], [\text{succ}(\text{succ}(0))], \ldots\},
\]

where \([\text{succ}^n(0)]\) is the congruence class of closed terms provably equal to \(\text{succ}^n(0)\). The function names are interpreted in the natural way on these sorts. For example, \(\text{succ}([n])\) is defined as \([\text{succ}(n)]\).

A particular computer implementation can be viewed as representing a concrete initial algebra of a specification, that is isomorphic to all other concrete initial algebras. The elements of the sorts in any initial algebra are called data elements, and the implementation computes data elements when they are needed.

We can now define “value” to mean the same as data element. To find out to which extent ADT’s fit our requirements for objects and to which extent they do not, we run through our shopping list. First, values have no identifier. We may say that a value is its own identifier, but this does not capture the idea that an identifier of an object is the unique substrate that remains invariant under all change.

The keywords in the second item on the shopping list are encapsulation, state and behavior. We may say that an ADT is a set of possible states and that operations represent state transitions. For example, a stack would be specified as the set of all its possible states. State-changing operations like \text{push} and \text{pop} would be defined as functions on this set. We will not view an ADT this way, because when we move from one state to the next, no identity information is preserved on which object changes state.

An ADT is a set of abstract values together with operations on those values, and in this set of abstract values there is no concept of an internal state. For example, the number 4 as abstract value (nor as concrete value, for that matter), has no internal state that remembers part of its past or determines part of its future. This is, among others, because the concepts of past and future have no meaning for values. This does not change at all if we use ADT’s to specify things we usually think of as having a dynamics, such as stacks or queues. Specified as an ADT, a stack is just a collection of values with operations on those values, and each of these values still has no internal state. Of course, the stack ADT as a whole may be viewed as a stack with each of its values as a possible state. This involves a change of meaning of the word “stack,” which is now used for the whole ADT (“the stack of natural numbers has 1, 2 and 3 pushed onto it”) instead of for each of the values in the ADT (“the stack 123”). There is nothing wrong with this, but it does represent a move from value-orientation to object-orientation. We will see below that this move requires the concept of a unique stack identifier that keeps different stacks apart and that represents the identity of a single stack through changes of state. We will argue below that addition of the concept of an object identifier is the first of the two steps which lead from values to objects, the second being the addition of the concept of an event.

Now, because there is no internal state to encapsulate, there is no semantic encapsulation in an ADT. However, at the syntactic level, the specification itself can be viewed as an encapsulation of a set of sort- and operation names such that they have limited scope. We can hide operation names so that they are invisible when the specification is imported into other specifications.
Values are classified into sorts, which have a taxonomic structure specified by \( \leq \). The inheritance of operations defined for a sort by all its subsorts is a semantic form of inheritance. There is also a syntactic form of inheritance, viz. the inheritance of a specification through the import mechanism.

Complex values can be built from simpler ones. For example, we can build tuples with atomic values and sets as components. Assume \texttt{NaturalNumbers} is extended to a specification \texttt{NaturalNumbersWithEq} that adds a function \( eq : NAT \times NAT \rightarrow BOOL \), that delivers \texttt{true} precisely when its two arguments denote equal elements. Also assume we have a specification \texttt{Sets}, parametrized by names \texttt{ITEM} and \( eq \). \texttt{Sets} specifies data elements of sort \texttt{SET}, that are finite sets of elements of sort \texttt{ITEM} and uses a Boolean function \( eq \) defined on \( ITEM \times ITEM \) to compare items. Specification of \texttt{Sets} is standard and is given in [45]. Then an example of how to build complex values follows:

```plaintext
value spec ComplexValues
  import
  NaturalNumbersWithEq,
  Sets using NaturalNumbersWithEq for Items
  binding [ ITEM \rightarrow NAT,
    eq \rightarrow eq]
  renaming [ SET \rightarrow NATS]
  sorts
    COMPLEX_VALUE
  functions
    (_,_) : NAT \times NATS \rightarrow COMPLEX_VALUE
end spec ComplexValues
```

Importing \texttt{Sets}, \texttt{ITEM} is bound to \texttt{NAT} and \( eq \) to the function with the same name in \texttt{NaturalNumbersWithEq}, and \texttt{SET} is renamed to \texttt{NATS}. \((\_,\_\)\) is a mixfix operator. Data elements of sort \texttt{COMPLEX_VALUE} have the form \((n, \{n_1, \ldots, n_k\})\). Other complex structures of values can be built just as easily.

Turning to the features of a specification language, the specification of new data types is the business of equational specification. Computational completeness is proven, for any computable data type is specifyable in an equational language that allows hiding of operation names [12]. This implies that for every recursive function on the natural numbers, there is an equational specification, possibly with hidden function symbols, that specifies this function. Overloading of function names is standard in equational specifications. This differs, incidentally, from redefining the implementation of a function (overriding), because there is no implementation specified.

The notion of querying an ADT has been, and is being studied extensively as equation solving and term rewriting. To query an ADT is to solve a system of equations in it, and then ask what the value of a number of terms is, with variables ranging over the solution set of the system. For example, we may ask what the value of the terms \( n, m, \) and \( n \ast m \), when \( n \) and \( m \) range over the solution set of

\[ n > 7 = \text{true} \]
\[ n + m = 10. \]

We may recognize the Select and Where clauses of SQL queries in this. Thus, most features of objects are already present in ADT's that are specified equationally. However, the concept of
object identifier, state and behavior are not present. We turn to these in the next sections, where I extend order-sorted equational logic to a language for the specification of conceptual models (CM's).

3 Specifying Objects

3.1 Attributes

3.1.1 Examples

The following example shows how we can specify objects with a local state identified by unique identifiers. First, we specify a simple ADT that contains object identifiers.

```
value spec PersonIdentifiers
  import  BooleanValues
  sorts  PERSON
  functions
    p0 : PERSON
    next : PERSON \rightarrow PERSON
    eq : PERSON x PERSON \rightarrow BOOL
  variables
    p,q : PERSON
  equations
    E1   eq(p0,p0) = true
    E2   eq(p0,next(p)) = false
    E3   eq(next(p),p0) = false
    E4   eq(next(p),next(q)) = eq(p,q)
end spec PersonIdentifiers
```

The sort PERSON contains infinitely many distinct oid's that can be compared for equality with the Boolean function eq. For two person oid's p1 and p2, eq(p1,p2) = true exactly when p1 = p2. This allows us to reason about the identity of persons. We extend this specification with a specification of person objects that have a local state, subject to integrity constraints.

```
object spec PersonObjects
  import  NaturalNumbers, AddressObjects,
  Sets using PersonIdentifiers for Items
  binding [ ITEM \rightarrow PERSON,
            eq \rightarrow eq]
  renaming [ SET \rightarrow PERSONS]
  attributes
    age : PERSON \rightarrow NAT
    address : PERSON \rightarrow ADDRESS
    children : PERSON \rightarrow PERSONS
```
variables
  \( p : \text{PERSON} \)

constraints
\[ \text{C1} \quad \text{age}(p) \leq 150 = \text{true} \]
end spec \( \text{PersonObjects} \)

An object specification can import value specification as well as other object specifications, and thereby makes the names declared in these specifications available, unless those names were declared hidden in those specifications. Some imported specifications may be parametrized and can be bound on import, such as \( \text{Sets} \). An object specification can declare attribute names, which are unary function names. We call them attribute names rather than function names, because the rules of interpretation differ for attribute- and function names. The argument sort of an attribute is called an identifier sort, because it contains the identifiers of objects as elements.

Unlike function names declared in value specifications, there are many possible functions that can be used as interpretations of attribute names declared in object specifications. Loosely speaking, each such interpretation is a possible world and represents one possible state of reality. Referring to figure 1, if \( p \) is a data element of the sort \( \text{PERSON} \) defined in \( \text{PersonIdentifiers} \), then the interpretation where we have \( \text{age}(p) = 30 \) represents a state of reality in which \( e \) is 30 years of age.

Attributes are subject to constraints, which are equations that must be true in every possible world. Thus, there can be no interpretation of the \( \text{age} \) function that has \( \text{age}(p) = 151 \). \( \text{C1} \) is an example of a local constraint, because it constrains the possible state of single person objects.

Possible worlds only differ in the interpretation they give to attributes names and agree on everything else. To point out the analogy with possible database states, one can think of each possible world as a set of tuples of the form \( (p, (\text{age} : n, \text{address} : a, \text{children} : \{p_1, \ldots, p_n\})) \), where \( p, p_1, \ldots, p_n \) are data elements of sort \( \text{PERSON} \), \( n \) of \( \text{NAT} \) and \( a \) of \( \text{ADDRESS} \). Attributes are then simply projection functions on these tuples. Note that attribute can have data elements, object identifiers, or sets of object identifiers as value.

I do not view the tuple \( (p, (\text{age} : n, \text{address} : a, \text{children} : \{p_1, \ldots, p_n\})) \) as an object, but as an object in a certain state. \( p \) is the identifier of the object whose possible states are represented by these tuples. These states are local, for they belong to \( p \) only. Object identifiers thus act as a principle of localization of state.

The following example specifies a more complex class of object. It assumes that there is a function \( \text{max} : \text{NAT} \times \text{NAT} \rightarrow \text{NAT} \) defined in \( \text{NaturalNumbers} \) that returns that argument that is larger than or equal to the other.

object spec ComplexPersonObjects
import
  PersonObjects
attributes
  ancestors : \text{PERSON} \rightarrow \text{PERSONS}
  max_age : \text{PERSONS} \rightarrow \text{NAT}
variables
  \( p, p', q : \text{PERSON} \)
  \( pp : \text{PERSONS} \)
constraints
\[ \text{C1} \quad p \in \text{children}(q) = \text{true} \rightarrow q \in \text{ancestors}(p) = \text{true} \]
C2 \[ p \in \text{ancestors}(p) \Rightarrow \text{ancestors}(p') \subseteq \text{ancestors}(p) = \text{true} \]

C3 \[ \text{max\_age}(\emptyset) = 0 \]

C4 \[ \text{max\_age}(pp \cup \{p\}) = \max(\text{max\_age}(pp), \text{age}(p)) \]

end object ComplexPersonObjects

In models of this specification, person objects have one more attribute, \textit{ancestors}. The infix operators \( \subseteq \) and \( \cup \) appearing in the equations are defined in the \textit{Sets} specification. C1 and C2 are positive conditional equations, which define the ancestors of \( p \) to be (C1) the parents and (C2) the ancestors of the parents. This is similar to the way it would be defined in deductive databases, except that the operational semantics uses term rewriting instead of resolution.

ComplexPersonObjects also defines objects whose identifier is a set of person identifiers. The states of these objects have the form \( (\{p_1, \ldots, p_k\}, \text{max\_age} : n) \) for a natural number \( n \). This is an object with identifier \( \{p_1, \ldots, p_n\} \) and attribute \text{mac\_age}, whose attribute value can be computed by means of constraints C3 and C4 from the attribute values of the objects of which it is "composed." Note, however, that the language in which we specify it is still first-order. Another example of an object with a structured identifier would be \textit{hire\_date} : \textit{PERSON} \times \textit{COMPANY} \rightarrow \textit{DATE}, which represents the date an employee is hired by a company. This is really a relationship attribute in ER terms.

3.1.2 Formalization

To formalize this, we need the concepts of conservative enrichment of a specification and reduct of a model. These are defined By Ehrig and Mahr [18]. A signature \( \text{Sig}_1 = ((S_1, \leq_1), F_1) \) is an extension of \( \text{Sig}_0 = ((S_0, \leq_0), F_0) \) if \( S_0 \subseteq S_1, \leq_0 \subseteq \leq_1 \), and \( F_0 \subseteq F_1 \). It is an enrichment if \( S_0 = S_1 \) and \( \leq_0 = \leq_1 \). \( \text{Spec}_1 = ((S_1, \leq_1), F_1, E_1) \) is an extension of \( \text{Spec}_0 = ((S_0, \leq_0), F_0, E_0) \) if the signature of \( \text{Spec}_1 \) is an extension of the signature of \( \text{Spec}_0 \) and \( E_0 \subseteq E_1 \).

Let \( \text{Spec}_1 \) be an extension of \( \text{Spec}_0 = ((S_0, \leq_0), \text{IF}_0, \text{IP}_0, E_0) \).

1. \( \text{Spec}_1 \) is a complete extension of \( \text{Spec}_0 \) iff for any \( s \in S_0 \) and any \( t \in T(\text{Spec}_1) \), there is a \( t' \in T(\text{Spec}_0) \), such that \( \text{Spec}_1 \vdash_{\text{Eq}} t = t' \).

2. \( \text{Spec}_1 \) is a consistent extension of \( \text{Spec}_0 \) iff for any \( s \in S_0 \) and any \( t_1, t_2 \in T(\text{Spec}_0) \), we have \( \text{Spec}_1 \vdash_{\text{Eq}} t_1 = t_2 \) iff \( \text{Spec}_0 \vdash_{\text{Eq}} t_1 = t_2 \).

3. \( \text{Spec}_1 \) is a conservative extension of \( \text{Spec}_0 \) iff it is a complete and consistent extension of \( \text{Spec}_0 \).

These concepts will be used as follows: We start from a fixed specification \( \text{Spec}_{\text{ADT}} \) of the ADT's we use in the CM specification, and enrich this specification conservatively with the declaration of functions called \textit{attributes}, and equations for them, so that we get the extended specification \( \text{Spec}_{\text{Stat}} \). Because \( \text{Spec}_{\text{Stat}} \) is an enrichment of \( \text{Spec}_{\text{ADT}} \), there are no new sort symbols introduced by it and their ordering is the same as in \( \text{Spec}_{\text{ADT}} \). Because it is a conservative enrichment of \( \text{Spec}_{\text{ADT}} \), it is a consistent enrichment, and this means that two closed \( \text{Spec}_{\text{ADT}} \)-terms are provably equal in \( \text{Spec}_{\text{Stat}} \) iff they are provably equal in \( \text{Spec}_{\text{ADT}} \) (I use the completeness of equational logic here [25]). Finally, because we use the initial semantics of ADT specifications, there is no unnecessary confusion between closed \( \text{Sig}_{\text{ADT}} \)-terms in any model of \( \text{Spec}_{\text{Stat}} \). This corresponds to Reiter's [42] \textit{unique name axioms}.

The second important property of initial algebras is that they contain no junk, which corresponds to Reiter's \textit{domain closure axiom}. If \( \text{ADT} \) is an initial model of \( \text{Spec}_{\text{ADT}} \), then all
Figure 2: Possible worlds and the underlying ADT.

data elements of $ADT$ are named by closed $Spec_{ADT}$-terms. So $ADT$ contains no junk. If $Spec_{ADT}$ is conservatively enriched to $Spec_{Stat}$, then $Spec_{Stat}$ is a complete enrichment of $Spec_{ADT}$, and this means that any model of $Spec_{Stat}$ containing $ADT$ as a $Spec_{ADT}$-reduct still contains no junk in the sorts of $ADT$. So we know, for example, that $age(p)$ is an element of $NAT$ that is already known in $Spec_{ADT}$. But because there are no other sorts in a model of an enrichment of $Spec_{ADT}$, a model of that enrichment containing $ADT$ as reduct, contains no junk at all. This corresponds to Reiter’s domain closure axiom.

We can now define $Spec_{Stat}$ more precisely. In all definitions, we assume a fixed ADT specification $Spec_{ADT} = (S, \leq, F, E)$ with an initial model $A_{DT}$.

**Definition 2**

1. A static object signature over $Spec_{ADT}$ is a conservative enrichment $((S, \leq), F \cup A, E \cup C)$ of $Spec_{ADT}$, where $F \cap A = \emptyset$ and $E \cap C = \emptyset$. The functions declared in $A$ are called attributes and the equations in $C$ are called static integrity constraints. Each domain of an attribute is called an identifier sort (this may be a tuple of sorts).

2. A possible world of $Spec_{Stat}$ is any model of $Spec_{Stat}$ that contains $ADT$ as a $Spec_{ADT}$-reduct. The intended model $PW$ of $Spec_{Stat}$ is the set of all the possible worlds of $Spec_{Stat}$.

Figure 2 clarifies the situation. $PW$ is a simple Kripke structure. Each of the black dots in $PW$ is a possible world, which contains $ADT$ as a reduct. The only differences between the worlds are the interpretations of the attribute names. All closed $Spec_{ADT}$-terms have the same interpretation in every possible world. In particular, the object identifiers (arguments to attributes) have the same denotation in every possible world. This formalizes the idea that the identifier of an object never changes.

At the textual level of specifications, attribute names, like other names, can be encapsulated in the specification that declares them. This is done by the hiding mechanism known from other equational specification languages [10, 23]. Thus, the specification $PersonObjects$ imports the specification $AddressObjects$, and the names in $AddressObjects$ that are not hidden, are available in $PersonObjects$. $PersonObjects$ may therefore add declarations of attributes of $ADDRESS$ and add constraints on $ADDRESS$ attributes. This is similar to defining person objects in stages, say $PersonObjects$ first and $ComplexPersonObjects$ later. Assuming $city$ is an attribute of $ADDRESS$ that is not hidden in $AddressObjects$, then we may also evaluate terms like $city(address(p))$ in $PersonObjects$. The names in this term are declared in two different specifications.

At the semantic level of objects in a model, it is not names that are encapsulated in a specification, but states that are encapsulated in objects. The state of an object is encapsulated in
that object because it stored as the attribute values of that object, and it is accessible by applying the attributes of the object to its identifier. Viewed this way, evaluation of \( \text{city}(\text{address}(p)) \) requires access to the state of two objects, a person object identified by \( p \) and an address object identified by the value of \( \text{address}(p) \).

This is independent of, and orthogonal to, the fact that the names in the term are declared in two specifications: \( \text{city}(\text{address}(p)) \) would still require access to two objects if all names in it would be declared in one specification. If a term like \( \text{city}(\text{address}(p)) \) is evaluated as part of evaluating a constraint (or query), then this evaluation is performed in a specification and is allowed if it is allowed by the import and hiding mechanisms. It is another matter whether one object has access to the state of another object. Regulating inter-object access requires an encapsulation discipline at the semantic level, in which objects are or are not prevented from looking into the state of other objects. This discipline is formalized as communication in the next subsection.

3.2 Object Updates and Messages

Objects can perform events, and events can do one or both of the following things: change the state of the object in whose life it occurs and/or access the state of other objects. Accessing the state of another object is called communication. I start with the specification of communication.

3.2.1 Communication

**Definition 3**

1. An event specification over \( \text{Spec}_{\text{ADT}} = ((S, \leq), F, E) \) is a conservative extension \( \text{Spec}_{\text{Event}} = ((S', \leq'), F \cup \text{OP}_{\text{Event}}, E \cup \text{E}_{\text{Event}}) \) of \( \text{Spec}_{\text{ADT}} \). \( S' \) introduces at least one new sort. One of the new sorts introduced in \( S' \) is called the sort of events. \( F \cap \text{OP}_{\text{Event}} = \emptyset \) and \( E \cap \text{E}_{\text{Event}} = \emptyset \).

2. Any algebra that contains \( \text{ADT} \) as a \( \text{Spec}_{\text{ADT}} \)-reduct can be chosen as intended \( \text{Spec}_{\text{Event}} \)-algebra. This algebra is called \( \text{EVENT} \).

Figure 3 clarifies the situation. \( \text{Spec}_{\text{Event}} \) declares at least one extra sort with respect to \( \text{Spec}_{\text{ADT}} \), which we will call \( \text{EVENT} \). It is an extension of \( \text{Spec}_{\text{ADT}} \), so we know that in the event algebra, events like \( \text{inc}_\text{salary}($1 + $1) \) and \( \text{inc}_\text{salary}($2) \) are equal. However, it is a conservative extension of \( \text{Spec}_{\text{ADT}} \), so no new identifications are added to those already present in \( \text{Spec}_{\text{ADT}} \) and this prevents \( \text{inc}_\text{salary}($1000) \) to be equal to \( \text{inc}_\text{salary}($1) \). Moreover, we know that whatever the intended event algebra is, it will contain \( \text{ADT} \) as \( \text{Spec}_{\text{ADT}} \)-reduct. So we know that \( \text{EVENT} \) does not add any data elements to sorts in \( \text{ADT} \). The advantage of this
construction is that we can experiment with different event algebras, as long as they satisfy this requirement. For example, if we extend the event algebra to a process algebra, we can experiment with different models of concurrency [45]. The following is an example event specification.

**event spec CommunicationAxioms**

**import**

SpecADT

**sorts**

FAIL ≤ EVENT

**functions**

fail : FAIL

& : EVENT × EVENT → EVENT [AC]

**variables**

e : EVENT

**equations**

E1 fail&e = fail

end spec CommunicationAxioms

The letters AC are meant as shorthand for the equations that define & to be an associative commutative binary operator. & will be used as communication operator below, to specify synchronous message passing between objects. fail is the failure event that will lead to no successor state. An attempt to execute an event synchronously with fail always fails. I assume the initial algebra semantics of this specification as intended event algebra. This satisfies the reduct requirement, because SpecADT also has the initial algebra requirement, but as said before, we can experiment with different algebras. In the next section, I give example specifications, which are formalized afterwards.

### 3.2.2 Example Specifications of Objects With Events

**object spec PersonDynamics**

**import**

PersonObjects, CommunicationAxioms

**events**

inc_age : PERSON → EVENT

change_address : PERSON × ADDRESS → EVENT

add_child : PERSON × PERSON → EVENT

**variables**

p, c : PERSON

cc : PERSONS

n : NAT

a : ADDRESS

**constraints**

C1 [inc_age(p)]age(p) = n + 1

C2 [change_address(p, a)] address(p) = a

C3 children(p) = cc → [add_child(p, c)] children(p) = cc ∪ {c}

end spec PersonDynamics
An event is just a function whose first argument sort is an oid sort, and whose result sort is the distinguished sort \( EVENT \) from \( Spec_{Event} \). (If the oid sort is a tuple of sorts, we regard this as one sort.) The interpretation of terms of sort \( EVENT \) will be that they are functions on \( PW \). Application of event \( e \) to a world \( w \) is called performing \( e \) in \( w \), and yields one or more possible next worlds (we allow nondeterminism in the general case).

The constraints contain the modal operator \([\cdot] \) from dynamic logic. The formula \([e] \phi \), with \( e \) an event term, is true in a world \( w \) iff \( \phi \) is true in the world(s) reachable from \( w \) by performing \( e \). So C1 says that an occurrence of \( inc\_age(p) \) necessarily leads to a world in which \( age(p) \) has been increased by 1. C2 similarly defines the effect of \( change\_address \) and C3 of \( add\_children \).

Using the inference rules given in table 1 below, it is easy to prove that C1 in \( Person\_Dynamic \) and C1 in \( Person\_Objects \) are jointly equivalent to

\[
\begin{align*}
\text{C1.1} & \quad \text{age}(p) = n \land n < 150 \rightarrow [inc\_age(p)] \text{age}(p) = n + 1 \\
\text{C1.2} & \quad \text{age}(p) \geq 150 \rightarrow [inc\_age(p)] \text{false}
\end{align*}
\]

Because \( \text{false} \) is true in no world, C1.2 gives a condition under which \( inc\_age(p) \) can lead to no world. C1.2 blocks any occurrence of \( inc\_age(p) \) in a world in which \( p \) is aged 150 or more.

The first argument of an event is the oid of the object in whose life the event occurs (I avoid the word "perform" in this sentence because the object may either initiate or suffer the event, cf. [30, 48, 41]). All constraints in \( Person\_Dynamics \) are local, for they contain only attribute applications to a single oid, and only event applications with the same oid as first argument. We allow only dynamic constraints that are local. Global dynamic constraints, which involve more objects, must be specified using communication between objects. The following example illustrates communication.

**object spec AccountObjects**

**import**

\( Person\_Dynamics, Communication\_Axioms, Natural\_Numberss \)

**sorts**

\( ACCOUNT \)

**attributes**

\( balance : ACCOUNT \rightarrow NAT \)

\( owner : ACCOUNT \rightarrow PERSON \)

**events**

\( add : ACCOUNT \times NAT \rightarrow EVENT \)

\( receive : ACCOUNT \times ACCOUNT \times NAT \rightarrow EVENT message \)

\( send : ACCOUNT \times ACCOUNT \times NAT \rightarrow EVENT message \)

**variables**

\( a, a_1, a_2 : ACCOUNT \)

\( n : NAT \)

**constraints**

\( C1 \quad balance(a) = b \rightarrow [add(a, n)]balance(a) = b + n \)

\( C2 \quad balance(a_1) = b \rightarrow [send(a_1, a_2, n)]balance(a_1) = b - n \)

\( C3 \quad balance(a_2) = b \rightarrow [receive(a_2, a_1, n)]balance(a_2) = b + n \)

**communications**

\( COM1 \quad send(a_1, a_2, n) \& receive(a_2, a_1, n) \)
All axioms of first-order logic

\[
\begin{array}{c|c|c|c}
\phi, \psi & \phi, \psi & \phi & \phi \\
\hline
\phi, \psi & \phi & \psi & [\text{MP}] \\
\hline
\forall x : (\phi) & [\text{G}] \\
\hline
[e] \phi & [\text{N}] \\
\hline
t = t & t_1 = t_2 \to t_2 = t_1 & (t_1 = t_2) \land (t_2 = t_3) \to (t_1 = t_3) & [\text{CON1}] \\
\hline
\{t_1/x\} = \{t_2/x\} & t_1 = t_2 & t_1 = t_2 & [\text{SUB1}] \\
\hline
[e_1] = [e_2] & [\text{e}\{~/e~\}] & [\text{e}\{x/e_2\}] & [\text{SUB2}] \\
\hline
[e] (\phi_1 \to \phi_2) & ([e] \phi_1 \to [e] \phi_2) & [\text{DL1}] \\
\end{array}
\]

Table 1: Inference rules for dynamic equational logic.

All names in the imported specifications that are not declared hidden, are available in the specification `AccountObjects`. However, the effect of events on objects must be defined using local constraints, as before. The events have as their first argument the object in whose life they occur. Two events are declared messages, which means that they cannot be performed in isolation. The constraints define the local effect of an event on the object in whose life it occurs, and the communications section extends the event declarations with a declaration of communication events. `send(a_1, a_2, n) & receive(a_2, a_1, n)` is a term of sort `EVENT`. It denotes an event consisting of the synchronous occurrence of `send(a_1, a_2, n)` and `receive(a_2, a_1, n)`, which models a transfer of money from `a_1` to `a_2`. The term `send(a_1, a_2, n) & receive(a_2, a_1, n)` differs from each of its component terms, and since it is itself not declared to be a message, nothing prevents it from occurring. And because its effect is the superposition of the effect of two events on different account objects, the money transfer is accomplished in a single step.

Note that we require that the communicating objects are different. If `a_1 = a_2` in `COM1`, the effect of the communication would be ill-defined.

### 3.2.3 Event Specification Syntax and Semantics

**Definition 4** Let `Sig_{Stat} = ((S, \leq), F)` be a static object signature and `Sig_{Event} = ((S', \leq'), F')` be an event signature, over a common ADT signature `Sig_{ADT}`, such that `F \cap F'` is the set of operation names declared in `Spec_{ADT}`. Then `Sig_{Dyn} = ((S', \leq'), F \cup F')` is called a dynamic object signature `Sig_{Dyn}` over `Sig_{Stat}` and `Sig_{Event}`. We write `Sig_{Dyn} = Sig_{Stat} + Sig_{Event}`.

Dynamic constraints are formulas in a language defined by the BNF

\[
\Phi ::= \phi \mid \phi_1 \land \ldots \land \phi_n \to \phi \mid [e] \phi_1 \land \ldots \land \phi_n \to [e] \phi_1' \land \ldots \land \phi_k'.
\]

where `\phi, \phi_1` and `\phi_j` are equations from the language of `Spec_{Stat}`, and `e` is a term of sort `EVENT`. I call this language dynamic equational logic. The inference rules for this language are given in table 1 [28, 41]. An inference rule has the form `\frac{H}{\Phi}`, where `H` is a set of formulas. If `H = \emptyset`, then `\Phi` is called an axiom. [CON1] and [SUB1] are the standard rules for congruence and substitution. `t\{t'/n\}` is the term obtained by substituting `t'` for every occurrence of `x` in `t`. This is allowed only if every sort that `x` has as a term is also a sort of `t'`. Assuming that the signature is regular, as we
do in this paper, every term has a unique least sort, and so the condition under which \( t' \) may be substituted for \( x \) is that the unique least sort of \( t' \) is less than or equal to the unique least sort of \( x \).

[SUB2] requires event terms that are equal in \( \mathcal{E} \), have equal effect on the possible worlds in \( \mathcal{P} \). [N] and [DL1], finally, are rules that hold in any Kripke structure.

The use of the inference rules is, of course, to prove properties of specifications. An example of this such a property is the condition under which an update inc_age is allowed, given the definition of the effect of inc_age and the static constraints it must respect. This condition is given in C1.1 and C1.2 above.

**Definition 5** Let \( \text{Spec}_{\text{Stat}} \) and \( \text{Spec}_{\text{Event}} \) be a static object specification and an event specification over \( \text{Spec}_{\text{ADT}} \), with signatures \( \text{Sig}_{\text{Stat}} \) and \( \text{Sig}_{\text{Event}} \). Then a dynamic object specification over \( \text{Spec}_{\text{Dyn}} \) is a conservative extension of both \( \text{Spec}_{\text{Stat}} \) and \( \text{Spec}_{\text{Event}} \) with signature \( \text{Sig}_{\text{Dyn}} = \text{Sig}_{\text{Stat}} + \text{Sig}_{\text{Event}} \).

**Definition 6** Let \( X \) be a set of variables, \( \Sigma = X \rightarrow (\cup_{s \in \text{SS}}) \cup \mathcal{E} \) be the set of all possible sort-preserving assignment to variables in \( X \), and \( T_{\mathcal{E}}(X) \) be the set of terms of sort \( \mathcal{E} \), containing variables from \( X \). Then a model of a specification \( \text{Spec}_{\text{Dyn}} \) is composed of a model \( \mathcal{P} \) of \( \text{Spec}_{\text{Stat}} \), a model \( \mathcal{E} \) of \( \text{Spec}_{\text{Event}} \), an algebra \( \text{ADT} \) of \( \text{Spec}_{\text{ADT}} \), such that all \( w \in \mathcal{P} \) and \( \mathcal{E} \) contain \( \text{ADT} \) as a \( \text{Spec}_{\text{ADT}} \)-reduct, plus a function

\[
\rho : T_{\mathcal{E}}(X) \rightarrow (\Sigma \rightarrow (\mathcal{P} \rightarrow \mathcal{P})),
\]

satisfying the requirement that

\[
\llbracket e_1 \rrbracket_{\mathcal{E}} = \llbracket e_2 \rrbracket_{\mathcal{E}} \text{ implies } \rho(e_1)(\sigma) = \rho(e_2)(\sigma).
\]

Figure 4 illustrates this. \( \rho(e)(\sigma) \) is a reachability relation defined for an event term and an assignment. For closed event terms, we write \( \rho(e) \). The requirement on \( \rho \) is the semantic counterpart of the inference rule [SUB2].

Truth of a formula \( [e] \phi \) in \( w \in \mathcal{P} \) under an assignment \( \sigma \) to variables is defined by

\[
w, \sigma \models [e] \phi \text{ iff for each } e \in T_{\mathcal{E}}(X) \text{ we have } \rho(e)(\sigma)(w) \models \phi.
\]

Truth of other formulas is defined in the standard way. With respect to this truth definition, the inference rules can easily be proven sound. Completeness can be proven relatively easily because what we have is a modal logic with several modal operators \( [\alpha] \) for different \( \alpha \). Inference rule [SUB2] then takes care of completeness. A sketch of a completeness proof is given in [41]. The complete proof is given in [49].
3.2.4 The Intended Model of Local Object Specifications

To express the idea of local state change, we limit ourselves to a subclass of all possible dynamic specifications. For these specifications, I will then define a distinguished, *intended model*. First, I distinguish two kinds of static constraints.

**Definition 7** An equation in $\text{SpecStat}$ is called a **local integrity constraint** if it is equivalent to a (possibly conditional) equation $\phi$ containing attribute applications, and if there is a variable $x$ such that all attribute applications in $\phi$ have the form $a(x)$. A constraint that is not local is called global.

$C_1$ in $\text{PersonObjects}$ and $C_3$ in $\text{ComplexPersonObjects}$ are local, the other constraints in $\text{ComplexPersonObjects}$ are global.

In the following definition, an event is called a **communication event** if it is equal, in $\text{EVENT}$, to a term with $\&$ in it, otherwise it is called an **atomic event**.

**Definition 8** Let $\text{SigDyn}$ be a dynamic object signature and $L(\text{SigDyn})$ be the dynamic equational logic formulas over $\text{SigDyn}$. A formula $\Phi \in L(\text{SigDyn})$ is called an **event constraint** if there is at least one event application in it. An event constraint is called **local** if

1. no events in $\Phi$ are communication events and
2. there is a variable $x$ such that each attribute application in $\Phi$ has the form $a(x)$ and each event application in it has the form $e(x, t_1, \ldots, t_k)$ for $k \geq 0$.

A local dynamic object specification is a dynamic object specification in which all event constraints are local.

A local event constraint can only change the attributes of a single object. All event constraints in $\text{PersonDynamics}$ and $\text{AccountObjects}$ are local. There will be no global event constraints in object specifications. So each event that is not a communication can only change the state of the object in whose life it occurs.

The intended effect of atomic events is defined by a **local frame assumption**, which says that if an event cannot be proven to change the value of an attribute, then the attribute remains unchanged under the event.

Let attribute $a$ have codomain $s$. If $\text{SpecDyn} \not\vdash a(x) = t \rightarrow \left[ e(x, t_1, \ldots, t_k) \right] a(x) = t'$, where $t$ and $t'$ are $\text{SpecADT}$-terms of sort $s$, then we assume $a(x) = t \rightarrow \left[ e(x, t_1, \ldots, t_k) \right] a(x) = t$. This is called the **local frame formula** of $e$ with respect to $a$.

In general, it is undecidable which assumptions to add under this local frame assumption, but in the particular examples shown, it is extremely easy. For example, the frame formulas of $\text{inc}_\text{age}$ in $\text{PersonDynamics}$ are

- $\text{address}(p) = a \rightarrow \left[ \text{inc}_\text{age}(p) \right] \text{address}(p) = a$ and
- $\text{children}(p) = c \rightarrow \left[ \text{inc}_\text{age}(p) \right] \text{children}(p) = c$.

Note that the local frame assumption preserves the reduct property of $\text{ADT}$, since it requires attribute values to be reducible to a $\text{SpecADT}$-term.
Given all local frame equations for all events, the effect of each atomic event on the attributes of the object in whose life it occurs is fully determined. Since in localized object specifications, an atomic event changes nothing else, the interpretation of atomic events in these specifications is fully determined.

The intended model of $Spec_{Dyn}$ can now be defined as follows.

1. Every term of the form $e(t_1, \ldots, t_k)$, with $e$ declared as message, is interpreted as fail.

2. Every communication event in which the first event arguments are not all interpreted as different oid's, is interpreted as fail.

3. If $\forall E \in \mathcal{T} \models e(\vec{i}) = e_1(x_1, \vec{t}_1) \& \cdots \& e_k(x_k, \vec{t}_k)$, then the effect of $e$ is the joint effect of $e_i$ on the attributes of $x_i$, $i = 1, \ldots, k$.

By 1, messages can only occur as part of communications. By 2, communications are always communications between different objects. Combining this with the localization requirement on specifications, the atomic events in a communication do not interfere, so the effect of $e$ defined in 3 is well-defined.

4 Discussion

4.1 Properties of Specifications

We already noted that the inference rules of table 1 can be used to prove properties of specifications. To prove properties of communication events requires extra axioms for $\&$. One standard axiom from dynamic logic is [28, 41]

$$[e_1] \phi \lor [e_2] \phi \rightarrow [e_1 \& e_2] \phi.$$  

If we add this to Communication Axioms and write $\text{transfer}(a_1, a_2, n)$ for $\text{send}(a_1, a_2, n) \& \text{receive}(a_2, a_1, n)$, then it is easy to prove that

$$\text{balance}(a_1) = b_1 \land \text{balance}(a_2) = b_2 \rightarrow [\text{transfer}(a_1, a_2, n)] \text{balance}(a_1) = b_1 - n \land \text{balance}(a_2) = b_2 + n.$$  

4.2 Existence

Existence of objects can be represented in a simple way by introducing a distinguished Boolean attribute $\text{exists}$, applicable to all oid's, whose value in a world is true for precisely the oid's that are considered to exist in that world.

Note that quantification is still over all possible objects. A constraint like $\text{exists}(p) \rightarrow \text{age}(p) \leq 150 = \text{true}$ explicitly quantifies only over existing objects. The extension of $\text{exists}$ in a world is in effect the set of oid's actually "stored" in that world. In what follows, I assume that all integrity constraints only quantify over existing objects and not over all possible objects. Thus, the condition $\text{exists}(x)$ for every variable $x$ in a constraint that ranges over oid's, is implicitly assumed.
4.3 Communication and Global Integrity Constraints

Suppose we declare an attribute \( \text{spouse} : \text{PERSON} \rightarrow \text{PERSON} \), and an event \( \text{get\_spouse} : \text{PERSON} \times \text{PERSON} \rightarrow \text{EVENT} \), subject to the constraints

\[
\begin{align*}
\text{GC1} & : \text{spouse}(\text{spouse}(p)) = p \\
\text{LC1} & : [\text{get\_spouse}(p_1, p_2)]\text{spouse}(p_1) = p_2.
\end{align*}
\]

\( \text{GC1} \) is a global constraint (definition 7) that can be violated if \( \text{get\_spouse} \) is performed. Violation can be prevented if \( \text{get\_spouse} \) is declared to be a message and if we define a communication

\[
\text{COM} \; \text{marry}(p_1, p_2) = \text{get\_spouse}(p_1, p_2) & \& \text{get\_spouse}(p_2, p_1).
\]

Thus, global constraints preservation must be realized by defining global communications. In general, a constraint linking two or more attribute values will require updates to these attributes to occur in parallel. If the attributes occur in the same object, this can be defined as a single local event, and if the attributes occur in different objects, this must be defined as a communication. This holds also for updating derived attributes, because a derivation rule expressing an attribute value in terms of a number of others is just a special kind of integrity constraint. The communication defining the simultaneous update to the related attributes is then a kind of update procedure as defined by Manchanda and Warren [37].

Further work needs to be done on this, both at the specification and at the operational level. For example, some global constraints require communications that are broadcast to a set of one or more objects that is not yet known at specification time. For example, in ComplexPersonObjects, \( \text{C5} \) links the state of several objects, and the state of any existing set \( pp \) of persons must be updated whenever the age of one of its elements changes. However, we do not know at specification time which sets of persons actually exist, and the syntax for communications require a specification of these sets. This problem can be solved by adding a broadcast mechanism to CMSL, which we plan to do in the near future.

Operationally, the problem just mentioned may be avoided by distinguishing derived from stored attributes and allowing updates of stored attributes only. However, this works only for global constraints that are derivation rules, and not for global constraints in general. Moreover, the problem of updating derived attributes would not be solved in that way. Research on operational aspects of updates will include this problem.

4.4 Object Identifiers

In CMSL, oid's are used to attach attribute values (i.e. they localize state) and events (i.e. they localize state change). They also serve to distinguish objects in the same state, and express persistence of object identity through change. Furthermore, they serve as the "address" that identifies communication partners. If an equational query (a set of equations) is solved in a possible world, then it is oid's that are returned as answers.

An important use of oid's is to mark the class(es) of the object identified. Oid's are just values declared to be of certain sorts and used in a certain way. This expresses the close relation there is between the identity of objects and their class. For example, a set of three passengers may well be a set of two persons, if one person in the set happens to be a passenger twice. Further work on
this is needed, though, because there are classes, called roles, which allow class migration [45]. For example, a person may migrate into and out of the class of passengers, and this would not be possible if the class of passengers would be a fixed subset of the class of persons.

5 Conclusion

In section 1, the list of requirements on object-oriented databases proposed in the object-oriented database Manifesto was split into two sublists, having to do with specification of object-oriented models and with the implementation of object-oriented databases, respectively. We saw in section 2 that order-sorted equational logic possesses most of the features required for a specification language of object-oriented models. The missing features were the ability to specify object identifiers, local object states, and object dynamics. These were added in the rest of the paper, by defining a language called CMSL that extends the order-sorted equational specification paradigm. Elsewhere [46], process algebra [7, 11] is added as well in order to define object life cycles. Normative constraints [40, 47, 51] and actors [39, 41, 48] can be defined in CMSL as well. As promised in the introduction, oid's play a central role in CMSL, among others to localize object states and state changes.

Current work on CMSL includes giving an operational semantics to updates. Some of the problems connected with this, such as updating derived attributes and preserving global integrity constraints, have already be mentioned. Work planned for the near future is giving an operational semantics to equational queries, which are sets of equations to be solved in a possible world.

Acknowledgements: Thanks are due to Frank Dignum, Reind van de Riet, and Hans Weigand for fruitful cooperation over the last few years. Paul Spruit eliminated some mistakes from a previous version of this paper. Special thanks are due to John-Jules Meyer for many discussions on constraints, objects and actors, which contributed significantly to the contents of this paper. Thanks are also due to the anonymous referees, who gave useful comments that helped to improve the quality of the paper.

References


