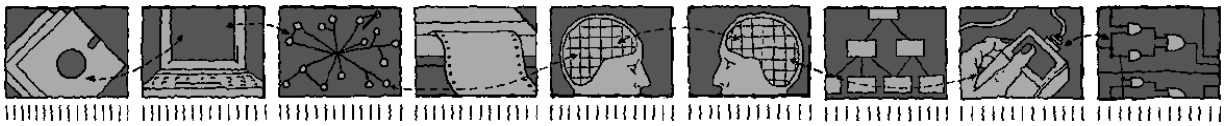
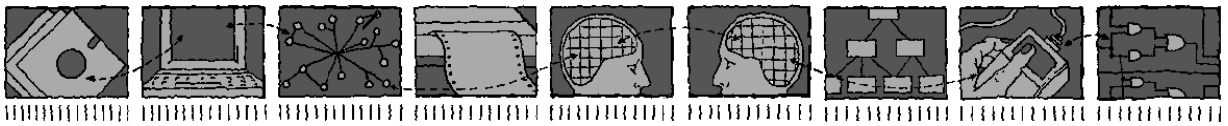


Department of Computing Science and Mathematics
University of Stirling



Formalizing OO Analysis with LOTOS

A.M.D. Moreira, P.B. Ladkin and R.G. Clark

Technical Report CSM-125

August 1994

Department of Computing Science and Mathematics
University of Stirling

Formalizing OO Analysis with LOTOS

A.M.D. Moreira, P.B. Ladkin and R.G. Clark

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email {Ana.Moreira || Robert.Clark}@compsci.stirling.ac.uk
|| Peter.Ladkin@loria.fr

Technical Report CSM-125

August 1994

Abstract

We define the denotational semantics of the concepts of Object-Oriented Analysis (OOA), in order to provide a generic description of the transformation from OOA into a formal model. We have developed the ROOA (Rigorous Object-Oriented Analysis) method, which builds on an object model created by using OOA methods and refines into a formal model expressed in LOTOS. We illustrate the semantics with ROOA-developed LOTOS specification.

Other semantics have focused on objects, and derived the meaning of classes and templates from them. In contrast, to fit in with OOA as it is practised, we focus on the ROOA concept of *class template*, and explain how the behaviour of objects in an implemented system is constrained by the behaviour expression contained in the template.

1 Introduction

The ROOA (Rigorous Object-Oriented Analysis) method has been developed to provide a formal requirements specification at the analysis stage of object-oriented system development. ROOA complements existing *Object Oriented Analysis* (OOA) methods (such as OMT [22], Coad and Yourdon [3] or Shlaer-Mellor [23]), enabling precision and formality in development where required, for example in safety-sensitive systems. ROOA starts from an object model built using one of the analysis methods and derives a formal specification in LOTOS which integrates the static, dynamic and functional properties of a system. A general explanation of the ROOA method and its results along with simple examples of development can be found [19, 20], while a discussion of how inheritance is handled may be found in [18].

The purpose of this paper is to specify the components of an object model and their relations in such a way as to illuminate the formal syntactic transformations used in a ROOA-like method. The results of ROOA are currently expressed in LOTOS, but the approach is general and could be used with other specification languages. One goal is to provide a simple-as-possible denotation of each concept so that practitioners of object-oriented analysis can get on with the hard job of building models without, say, having to learn category theory, universal algebra or fixpoint theory; requiring LOTOS is already enough! The actual formalisation of the concepts is thus of secondary importance, accomplished where possible simply by listing components as tuples, functions or relations by giving their denotational type (i.e. the domain and range sets). Nevertheless, there are technical problems arising with even so straightforward an approach. Our main concern is to identify and solve these problems. The main problem solved here is: how the behaviour of objects (in the system) is derived from the expressions of generic behaviour contained in the class templates in the model. The interpretation of inheritance is handled elsewhere [18]. We assume some familiarity with the standard LOTOS language [10], as found for example in [2, 26], and also with the Z notation [25].

Object-oriented system development views a system as a collection of interacting objects. First, an *object model* is built in the OOA phase. The result of the analysis phase is a requirements specification. The object model specifies the *class templates*, which include specification of the *attributes* (properties) of, and the *services* offered by, objects in the class defined by the template. Objects, which exist in the system proper and are not present in the object model, are *instances* of some class template.

The ROOA object model also specifies which services are offered and used, and the generic *behaviour* of objects using services. The services offered by objects are used by objects in a different class. The channels corresponding to these uses of services are called *message connections* and are specified statically in the object model using various devices such as an *Object Communications Diagram* and *Object Communication Table* in ROOA. When modelling a system, we abstract from its concrete objects, and describe it in terms of its class templates. A class template describes the static and dynamic (behavioural) properties of objects of the same kind. Our main focus here is to explain how the behaviour of objects at run-time is specified through the generic behaviour description given in the class template.

Motivation For This Work. One motivation — precision — for a formalisation of object modelling concepts and their expression has already been mentioned. We motivate the approach taken in ROOA and its semantics by contrasting the goals of ROOA with those of other work.

Another motivation is enhanced expressive power. Object models suggested in the OOA literature are rendered less expressive by their dependence on particularly simple behavioural models (e.g. finite automata [3, 22]) and particular notions of atomic action. The semantics of LOTOS, used for ROOA, allows much richer expression of behaviour than these proposed models.

Other approaches to giving formal structure to OO concepts have started with objects as fundamental and derived other concepts from these. They are mainly concerned with object oriented *programming* or with object-oriented *design* (OOD [4, 22, 24]), not with OOA. Examples include a formalisation [1] using Lamport's TLA logic [14], and formalisations using universal algebra and category theory [6, 7, 8]. These approaches are as powerful as LOTOS for describing the behaviour of objects. However, the analysis stage of development gives primacy to class templates and their meaning, and objects are simply not around. Hence these approaches do not appear to explain the concepts of OOA in practical methods such as OMT as ROOA does, in a way understandable by practitioners without advanced mathematical experience.

Furthermore, these other approaches do not separate static from dynamic structure — architecture from behaviour — in the way emphasised by a well-built object model [22], or by some approaches to Open Distributed Processing [9]. For example, technical problems of distributed actions such as recovery from failure (e.g. [17]) are not properly solved in the analysis stage. The philosophy of OOA suggests that they should be addressed at the design stage, using methodologies developed for this purpose. According to this philosophy, such classes of problems should be separated from the architecture of the system as expressed by the static model, as in ROOA. It is not clear to us how one can achieve this separation easily with approaches such as [1, 6, 7, 8] which are focused on objects.

Other approaches to development of formal system specifications in an object-oriented manner may be found in [5, 12, 16]. These approaches introduce new methodology in order to obtain formal specifications, and it is not clear that they build on practical approaches such as [11, 22], as in ROOA. The goal of [5] is to integrate OO concepts into LOTOS specifications (rather than the other way around, as in ROOA), and "the immediate area of application ... is the development of international standards for ... Open Distributed Processing (ODP) systems". A similar goal with respect to VDM-style development underlies the work of [16]. Finally, [12] is concerned with a formal notation for object-oriented programming and design, not with OOA. Because of their goals, these approaches focus on the specification of objects rather than the concepts of OOA, and do not strictly adhere to the division between analysis and design recommended by [3, 11, 22, 23] and employed in ROOA.

2 Basic Concepts: Set of Values and Variables

First we define the set *set_of_values* of all possible sets of values of data types. The extension of any data type is in *set_of_values*. We also define an infinite set of *variables*. Specification languages have various ways to handle data typing, some extensional (data types are identical if their sets of possible values are identical, no matter what the names) or intensional (not extensional). We only need to use data values (i.e. extensional data typing) for this work, but a language such as LOTOS does not have fully extensional data typing. We must therefore be flexible enough in what we do to cohere with the data typing principles we might encounter in this and other languages. We define $\mathcal{V}$ as the set containing all data values, and *data types* as the set of data types (whatever they may be). We define a separate naming function *extension* which associates a data type with a set of values, and do not say (because we do not need to) what the properties of this function are, or how identities between data types are handled. Our development ensures that the sets *class_templates*, *objects*, *attributes*, *services*, *data types*, *message types* are all disjoint.

We further require that the two sets

$$set\_of\_values = \{v \mid v \subseteq \mathcal{V}\} = \mathbb{P}\,\mathcal{V}$$
$$variables = \{x_i \mid i \in \mathbb{N}\}$$

are disjoint from each other and from everything else in sight. We define a naming relation

$$named\_by : STRING \times (variables \cup data\ \_types \cup class\ \_templates \cup objects \cup$$
$$\cup\ attributes \cup services \cup message\ \_types \cup object\ \_generators)$$

which assigns names to everything in sight. We also have a mapping

$$extension : data\_types \rightarrow set\ \_of\ \_values$$

which gives the extension of each data type. Finally the set

$$variable\_type\_pairs = \{\langle x, d\rangle \mid x \in variables, d \in data\ \_types\}$$

allows binding of variables in a particular context with data types.

The justification for these definitions is as follows. We accumulate all possible set-of-values values in $\mathcal{V}$, and therefore a specific data type will have an extension which is a specific subset of the values of $\mathcal{V}$. We need an infinite set of variables for standard syntactic reasons. Variables in a particular use are generally bound to data types: we want to be able to select a variable x of specific type *real*, say, in a given application. Therefore we need a set of variable-to-data-type bindings.

3 Defining the Concepts of OO Models

We define the various concepts *class template*, *attribute* and *service* in the static structure. We may define from these concepts the derivative notions of *object*, and *class* which are not in the object model, but in the implemented system. An object is an instance of a class template, which is to say that it has the attributes specified in the template, the services provided therein, and the behaviour defined therein. The notion of embodiment is thus quite simple: since everything is defined in the template except the identity of particular objects (although their sort is defined), an object is defined by assigning an identifier, and assigning behaviour by instantiating a free variable in the class template to the object identifier. A class is defined as the set of all objects instantiated from a given class template. In ROOA we define the instantiation of objects from class templates by means of a artifact we call an *object generator*.

Suppose the item *item* is the name of an element of a tuple (or sequence or record) defining *concept*. We shall in general write *concept.item* to refer to the value of this *item* in *concept*, as one normally does for records.

3.1 Class Template

A class template is defined by a *name*, a finite set of attributes Σ_A, a finite set of services Σ_S and a behaviour description $\mathcal{B}$ which apply to all objects instantiating this template. Thus, we define:

$$class\_template = \langle name, \Sigma_A, obj\_id, \Sigma_S, \mathcal{I}, \mathcal{B}\rangle$$

where $name \in dom(named\_by \rhd class\ \_templates)$ is the name of the class template [1]. There is a distinguished attribute $obj\ \_id \in \Sigma_A$, known as the object-identifier attribute. Values of $obj\ \_id$ are the possible identifiers of the different objects instantiated from $\mathcal{B}$. The function $\mathcal{I} : \mathcal{S}_{c_t} \rightarrow \mathbb{P}\,\Sigma_S$, describes the services that are available for each given state of an instantiated object, i.e. the

[1] For definition of the domain and range restriction operators $\lhd$ and $\rhd$, as well as the domain and range selectors *dom* and *ran*, see [25, pages 96, 98]

interface of an object at a given state. This function is explained in Section 3.3. $\mathcal{B}$ describes the typical behaviour of objects instantiated from the class template, and is defined by means of LOTOS constructs. We thus commit ourselves here to a specific form for $\mathcal{B}$. It is, however, important that no matter what language $\mathcal{B}$ is formulated in, it has certain properties which may be expressed using concepts from the syntax of logic. (The function of $\mathcal{B}$ is explained in Section 3.2.) The purpose of this requirement is that a class template c_t defines the behaviour of arbitrary objects instantiating c_t, and mentions no specific object by name. It may also refer to objects instantiating other templates, including the services they offer, again without mentioning specific objects by name. This is accomplished in logical syntax simply by using free variables.

The syntactic details of how this is accomplished in particular behaviour description languages such as LOTOS, SDL, Manna-Pnueli temporal logic, or TLA, are dependent on these particular languages. Therefore we explain such syntactic restrictions here generically, but illustrate with the ROOA derivation in LOTOS. Individual languages will require that the restrictions are translated into the syntactic restrictions that make sense for these languages.

3.2 The Generic Behaviour Description $c_t.\mathcal{B}$

The behavioural description $c_t.\mathcal{B}$ contained in a definition of c_t may have different forms, depending on the description language chosen. In ROOA, it is part of a LOTOS specification, usually a LOTOS process definition. But a semantic explanation of $c_t.\mathcal{B}$ should also explain its function if expressed in another description method such as SDL or TLA. We explain it here and in Section 4.3.

The behavioural description $c_t.\mathcal{B}$ is *generic*, in that it should provide a schema for describing the behaviour of an arbitrary object instantiated from c_t. We describe features we require of $c_t.\mathcal{B}$, so that it may be transformed into a description of the behaviour of an individual object instantiated from c_t. The procedure for transformation itself is described in Section 4.3.

We describe the structure of $c_t.\mathcal{B}$ in general syntactic terms taken from logic, using the notions of (logical) *individual variable* [2], and *substitution*, because these have been well worked out and understood for a century. We need to interpret these notions in a given target language (LOTOS, SDL, TLA, *etc.*). We illustrate with a LOTOS example in ROOA.

Since $c_t.\mathcal{B}$ is intended to be a generic behaviour expression, it must contain terms which are not constant, but which take different values when used to describe the behaviour of a particular object. (In logic, these terms are said to *range over* their collection of values.) To obtain generality of reference over individuals [3], logic employs individual variables ranging over individuals of a given sort. There is no *a priori* bound on the number of objects instantiated in a system from the class templates in the object model. We require a set Y of individual variables ranging over values of the attribute obj_id (that is, over the set $dom(named\_by \rhd objects)$). We also use the set of terms C referring to the class templates. There is a fixed set of class templates in the model, namely, the set $ran(named\_by \rhd class\_templates)$, and the set of their names is $C = dom(named\_by \rhd class\_templates)$.

When a variable $y \in Y$ is used in the behaviour expression $c_t.\mathcal{B}$ to refer generically to an object, it must range over objects instantiated from a particular class template c. We denote this binding by $c.y$. We also require a distinguished individual variable ι, used in $c_t.\mathcal{B}$ in the form $c_t.\iota$. When $c_t.\mathcal{B}$ is used to describe the behaviour of a particular object $\mathtt{ob}$ the value of whose obj_id is oid, the term $c_t.\iota$ will be replaced by oid (see Section 4.3). Thus $c_t.\iota$ is used to refer to the *'currently instantiated object'*. This corresponds to the concept called *'self'* in Smalltalk, or *'this'* in C++. See Section 4.1 for definition of the concept *object*, and Section 4.3 for an explanation of how the behaviour of the object $\mathtt{ob}$ is obtained from the expression $c_t.\mathcal{B}$.

When $c_t.\mathcal{B}$ mentions a service of an object instantiated from a different class template c, we write the occurrence as $c.y.service.name$, where $c \in C$, $y \in Y$, and $service \in c.\Sigma_S$ (the set of services offered by template c). Similarly, when an attribute of an object instantiated from c_t is referred to (only attributes of objects instantiating the current class template may be referred to

[2] This notion should of course be distinguished from the notion of program variable, or *variable* as used in LOTOS.

[3] We also employ the term *individual* for a given 'thing', to avoid confusion with the technical term *object*.

— all values of attributes of objects instantiating other templates must be accessed by services offered by those objects) then a term of the form $c_{\ t}.y.attribute.name$ is used.

In order to explain how the generic behaviour expression $c_{\ t}.\mathcal{B}$ is used to generate an expression describing the behaviour of a particular object in the system with identifier *oid*, we must identify the terms of the form $c_{\ t}.\iota$, $c.y$, $c.y.service.name$ and $c_{\ t}.y.attribute.name$ and all occurrences of them in $c_{\ t}.\mathcal{B}$.

3.3 The Visibility Function $\mathcal{I}$

Objects have 'state' [22]. We define the *value* of an object at any point to be the collection of values of the attributes of the object. We identify the *state* of the object at a given time with its value. In any particular state, not all of the services may be offered by the object. $\mathcal{I}$ is the *visibility* function, that for each state of an object tells which services are visible in that state. In the banking system problem in Section 3.4, for example, in an *account* object, a *withdraw* service may not be offered if the value of *balance* is 0. With each state of the object we may associate the set of services which are offered in that state. However, which services are offered in a particular state is (a) dependent on properties of the state of that object alone; and (b) invariant over all objects instantiating the same template. For example, the fact that *withdraw* is not offered in a state in which *balance* is 0 is a generic constraint on all objects of type *account*. Thus, visibility is statically determined: the collection of all such constraints, the *visibility* function, is a feature of the class template. Where $\mathcal{S}_{\ c_t}$ is the set of states (defined below), $\mathcal{I}$ has the type $\mathcal{I} : \mathcal{S}_{\ c_t} \rightarrow \mathbb{P}\,\Sigma_S$.

Since a state of an object is its value, the set of attribute-value pairs of the object at a particular time, we require that the set of states $\mathcal{S}_{\ c_t}$ is the collection of all possible combinations of values of all attributes. But not all of these sets of attribute-value pairs may be attained by some object in some run of some system, in other words, these potential states may not be *reachable*. However, it is often combinatorially intractable to determine which states are reachable [21]. Furthermore, it is evident from reachability analyses that the distinction between those states which are reachable or not is properly a logical consequence of the behavioural description of the system, and thus of logical properties of the behavioural descriptions $c.\mathcal{B}$ for all class templates c. Therefore the distinction between reachable and non-reachable states should not be made in the denotational description of the class template. Thus, we may define $\mathcal{S}_{\ c_t}$ to be the collection of all assignments of values of the right type to attributes in $\Sigma_{\ A}$, where $a.value\_set$ is the value domain of the attribute (see 3.5):

$$\mathcal{S}_{c_t} = \{f \mid f : \Sigma_A \rightarrow \mathcal{V} \wedge \forall\, a \in \Sigma_A \bullet f(a) \in a.value\_set\}$$

where $value\_set \in data\_types$.

3.4 Class Template in LOTOS

Consider a banking system where accounts can be debited, credited and queried for their current balance. These operations can either be performed by the counter tellers or by a customer using a card in the automatic teller machines.

ROOA derives a LOTOS process definition for a class template `Account` from an analysis of an object model involving a banking system as follows. This example is a modified version of one in [20], and is explained below:

```
process Account[a](this_account: State_Account) : noexit :=
 ( a !deposit !Get_Account_Number(this_account) ?m: Money;
   exit(Credit_Account(this_account, m))
 []
   a !get_balance !Get_Account_Number(this_account) !Get_Balance(this_account);
   exit(this_account)
 []
   ( choice zero_balance: Bool []
```

```
      [not zero_balance] ->
      ( a !withdraw !Get_Account_Number(this_account) ?m: Money;
        ( choice enough_money: Bool []
          [enough_money] ->
            a !rtn_withdraw !Get_Account_Number(this_account) !true;
            exit(Debit_Account(this_account, m))
        []
          [not (enough_money)] ->
            a !rtn_withdraw !Get_Account_Number(this_account) !false;
            exit(this_account)
        )
      )
      [zero_balance] -> exit(this_account)
    )
  ) >> accept update_account: State_Account in Account[a](update_account)
endproc
```

We explain how to identify the various syntactic components of the object definition in the LOTOS expression informally. It should be clear that the identifications have syntactic definitions, but it would be merely tedious to write these definitions formally.

The name of the class template is `Account`; the set of attributes Σ_A, including the *obj_id*, are given in the ADT where the sort `State_Account` is defined; the set Σ_S of services offered are defined as the first *action denotations* [26] of the choice operator '[]'. Specifically, the three services offered are defined by the expressions:

```
a!deposit !Get_Account_Number(this_account) ?m: Money;

a!get_balance !Get_Account_Number(this_account) !Get_Balance(this_account);

a!withdraw !Get_Account_Number(this_account) ?m: Money;
```

where `Get_Account_Number(this_account)` is an operation specified in the ADT where the sort `State_Account` is defined, returning the object identifier of an object instantiated from the class template `Account`. It corresponds to the individual variable $c_t.\iota$ reserved for the *'current object'*. Σ_S is the set of services whose names are `deposit`, `get balance` and `withdraw`. Specifically,

$$\Sigma_S = ran(\{\texttt{deposit}, \texttt{get\_balance}, \texttt{withdraw}\} \lhd named\_by)$$

Given these bindings of syntactic parts of the LOTOS expression to the formal parts of the class-template tuple and the identification of expressions of the form $c_t.\iota$, $c.y$, $c.y.service.name$ and $c_t.y.attribute.name$, the behaviour description $\mathcal{B}$ is given by the body of the process definition.

The interface function $\mathcal{I}$ is defined in LOTOS by identifying guarded expressions in the behaviour description. Consider, for example, the expression:

```
[not zero_balance] -> a !withdraw !Get_Account_Number ...
 ...
[zero_balance] -> exit(this_account)
```

This ensures that a client may only use the service `withdraw` if the balance is not zero. As ROOA deals with the analysis phase, a reference to the state of an object may be a symbolic reference. Therefore, in process `Account` we use the generalised LOTOS choice operator `choice` to cover the two possible situations expressed by the two guards `zero balance` and `not zero balance`, whose values depend on the state of the object when the guard is evaluated at run time (see [19, 20] for details).

The gate `a` is used to represent the channel of communication between `Account` and `Counter_Teller`.

The argument `this_account` defines the state of objects of class template `Account`, that is, it defines Σ_A and the collection of values at any time. In ROOA, this is defined by an ADT, as follows:

```
type Account_Type is Account_Number_Set_Type, Money_Type, Balance_Type
sorts State_Account
 opns Make_Account   : Account_Number, Balance -> State_Account
      Credit_Account : State_Account, Money    -> State_Account
      Debit_Account  : State_Account, Money    -> State_Account
      Get_Balance    : State_Account           -> Balance
      Get_Account_Number : State_Account       -> Account_Number
      ...
 eqns forall a: State_Account, n: Account_Number, m: Money
   ofsort Balance
      Get_Balance(a)  = Some_Balance;
   ofsort Account_Number
      Get_Account_Number(Make_Account(n,m))   = n;
      Get_Account_Number(Credit_Account(a,m)) = Get_Account_Number(a);
      Get_Account_Number(Debit_Account(a,m))  = Get_Account_Number(a);
endtype
```

There is in each such ADT a distinguished operation always prefixed with the term `Make` which has Σ_A as domain type and returns the sort of the object. It is used by the *object generator* (Section 3.7) to instantiate objects. The object generator is an artifact of ROOA. Σ_A may be read directly off the domain expression in the LOTOS-type of the `Make` *whatever* operation. The ADT `Account_Type` shows the attributes in Σ_A as `Account_Number` and `Balance`. The operations defined in `Account_Type` are components of the methods used to perform the services offered (see Section 3.6).

3.5 Attributes

We define an attribute as a tuple

$$attribute = \langle name, value\_set \rangle$$

where $name \in dom(named\_by \rhd attributes)$ and $value\_set \in set\_of\_values$ is the set of all values that the attribute may have.

In ROOA, an attribute is defined in a LOTOS ADT. In the banking example, we saw that the class template `Account` has two attributes: `Account Number` and `Balance`. These attributes appear as arguments of the operation `Make Account`.

3.6 Services

A service $s_i \in c_t.\Sigma_S$ is offered by all objects instantiating a given class template c_t. It is used by other objects to query or change the state of the object which offers it. Objects which are instances of the same class template may not directly use each others' services. (If an object A uses a service of object B, we say that objects A and B *communicate*. The communication channels between objects are determined statically, in the model, and correspond to the mysterious gate `a` in the previous example.) Each service is defined by a unique name and by one or more methods. We define its type as follows:

$$service = signature \frown \langle method_1 ... method_n \rangle$$
$$signature = \langle name, obj\_id, in\_parameters, out\_parameters \rangle$$
$$in\_parameters \in attributes^{\,n} \times variable\_type\_pairs^{\,m} \times D$$

where $\frown$ is the concatenation operator on sequences, $D \in data\_types^{\,p}$ is a p-tuple of specific data types; and

$$out\_parameters \in Q$$

where $Q \in data\_types^{\,q}$; also $name \in dom(named\_by \rhd services)$.

We distinguish here the static structure of a method (simply its signature with its input and output parameters) from the description of how the method is used or how the value returned in the output parameters is calculated. The use of a method properly belongs to the dynamic description, as does the description of how the value is calculated, and must be described in $\mathcal{B}$. (TLA [13] *actions* correspond to our methods.) Therefore, these features belong to the LOTOS description of the behaviour of the service. A *method* is defined by a LOTOS behaviour expression such as

$$method_i = \alpha;\ exit(ADT\_op(...))$$

where α is a collection of action denotations which can include invocation of services defined in other objects, for example. The *exit* construct indicates successful termination. This construct may have arguments, which we use, in this case, to change the value of one or more attributes of the object, by invoking an operation defined in an ADT. This operation has the form:

$$ADT\_op : state\_sort, variable\_type\_pairs \xrightarrow{r} state\_sort$$

where *state_sort* represents the sort of the states of an object, e.g. `State Account` in the ADT above.

The behaviour of the method which specifies the service `withdraw` in the banking system, is given by the two following behaviour expressions:

```
( choice enough_money: Bool []
  [enough_money] ->
     a !rtn_withdraw !Get_Account_Number(this_account) !true;
     exit(Debit_Account(this_account, m))
[]
  [not (enough_money)] ->
     a !rtn_withdraw !Get_Account_Number(this_account) !false;
     exit(this_account)
)
```

where the *ADT_op* is `Debit_Account`. Each one of these two behaviour expressions define one method, each of which is used as an alternative of the other. Notice that there are methods which change the state of an object (called *modifiers*) and others which do not (called *selectors*). The selectors only return the value of one or more attributes, as happens with `Get Balance`. In any case the *exit* construct always returns the state of the object, whether updated (as in `exit(Debit_Account(this_account, m))`) or not (as in `exit(this_account)`).

3.7 Object Generator

In ROOA, an object is instantiated from the class template by a special mechanism called an *object generator*. Although an object generator knows about the objects (identifiers) already created, it is mainly used to instantiate a class template.

$$object\_generator = \langle name, c\_t, create\_obj\_c\_t, object\_id\_set, \beta \rangle$$

An object generator has a name, $name \in dom(named\_by \rhd object\_generators)$, and instantiates objects from the template by offering the service $create\_obj\_c\_t$ in a manner described by the behaviour β. It also contains a distinguished name *object_id_set* for the set of identifiers of objects already instantiated from c_t.

When an object is instantiated by a call to $object\_generator.create\_obj\_c\_t$, it has an initial state. The initial state is defined in the behavioural description β. A behavioural description without a specified initial state is an invalid description. Most behavioural specification languages such as LOTOS, SDL and TLA either include or require such an initial state definition.

An object generator for class template `Account` looks as follows:

```
process Accounts[a](accs: Account_Number_Set) : noexit :=
  ( a !create ?acc_counter: Account_Number [(acc_counter notin accs)];
    ( Account[a](Make_Account(acc_counter, 0 of Balance))
    |||
      Accounts[a](Insert(acc_counter, accs))
    )
  )
```

where `Accounts` is the name of the object generator and `accs` is the set of identifiers of objects already instantiated from class template `Account`. The service

```
a !create ?acc_counter: Account_Number [(acc_counter notin accs)];
```

corresponds to *create_obj* $\mathcal{L}_t$ and it uses value generation to generate an object identifier of sort `Account_Number` which is then used to instantiate the class template by executing

```
Account[a](Make_Account(acc_counter, 0 of Balance))
```

The operation `Make_Account` is defined in the ADT `Account Type` and it creates a value of the sort `State_Account`. The initial state of the object account created is the result given by `Make_Account`.

The behaviour β is given by the body of the process definition. The object generator is defined recursively so that we can continue to create objects.

4 Defining the Concepts of OO Systems

Certain concepts of OO methods are not properly part of the object model, as described for example in [22], but are part of the system design that derives from a given object model. Most attempts to define the semantics of OO systems concentrate on objects, classes, inheritance and aggregation, and are properly part of the system design, not the model. We believe our approach fits more closely the ontology of object-oriented models, and it coheres with the powerful behavioural description methods in LOTOS.

4.1 Object

An object in the design represents what 'runs' in the implemented system. A class template is a definition which is used to create objects. An object is an instantiation of a class template. The class template includes the description of the generic behaviour of an object, as well as defining all attributes which the object may have. In order to know all about an object it is sufficient to know about only which template c_t it was instantiated from, and its identifier (the value of the attribute $c_t.obj\_id$). The object's behaviour must be derived from $c_t.\mathcal{B}$. Hence we define

$$object = \langle ident, class\_template.name \rangle$$

where $ident \in dom(named\_by \rhd objects)$.

4.2 Behavioural Constraints vs. Behavioural History

In the definition of *object* in Section 4.1, the object itself is rather bare. Objects have a history of behaviour, which may be different for each object, and furthermore is not derivable from the class template. Where, one may ask, is this behavioural history in the definition of object?

The answer is that the history of behaviour of an object, no matter how important for the operation of the object in the system, is not properly part of a *semantic* account of class template, object, and their relation. In contrast, an account of how the *constraints* on an object's possible behaviour are obtained *is* properly part of the semantics. $c_t.\mathcal{B}$ is a constraint expression, with

lots of free variables. We describe in Section 4.3 how the behavioural constraints on an object are obtained from the constraint expression in the template.

This position may be compared with that of semantics of programs. Compare objects with programs. A specification includes the constraints on a program's behaviour. An actual behaviour of the program, or a partial behaviour (a *history*) is described by a (partial) trace. The specification describes the set of all possible traces. Each individual partial trace that satisfies the specification, and which may or may not occur during the lifetime of the system, is not considered part of semantics. It is up to the program whether it wants to keep around such information about its previous behaviour. In many object-oriented systems, this may be important for auditing, just as in previous designs in the before object-orientation era, keeping history variables around for the same purposes was also considered important. But it is not part of semantics. It is part of programming.

4.3 Deriving the Behaviour of an Object from the Class Template

Let ob be the *'current object'* and let its identifier ob.*ident* be *oid* for notational simplicity. The behaviour of ob can be described by $\mathcal{B}[c_{\ t}.\iota/oid]$. Where $\mathcal{B}[c_{\ t}.\iota/oid]$ is the expression obtained by substituting *oid* for every occurrence of the distinguished individual variable $c_{\ t}.\iota$ in $\mathcal{B}$. $\mathcal{B}[c_t.\iota/oid]$ still contains individual variables, say for example simply $c.y$, for other objects whose services ob may use. Since the expression still has an unbound individual variable $c.y$, it does not yet describe concrete behaviour. The description is supposed to say what behaviour is allowed between ob and other objects instantiated from c.

Suppose that $\mathtt{ob}_1$ and $\mathtt{ob}_2$ are precisely the instantiated objects from template c, with identifiers $\mathtt{ob}_1.ident$ and $\mathtt{ob}_2.ident$ respectively, which we denote $c.oid_{\ 1}$ and $c.oid_2$ for simplicity. The required description of ob's interaction with one specific other object is obtained when any value of another object identifier instantiating a template c is substituted for the variable $c.y$ appearing in $\mathcal{B}[c_t.\iota/oid]$. The object ob may engage in its c-object interactions with $\mathtt{ob}_{\ 1}$, and also with $\mathtt{ob}_{\ 2}$. Thus the two expressions $\mathcal{B}[c_{\ t}.\iota/oid][c.y/c.oid_{\ 1}]$ and $\mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 2}]$ describe the interactions in which ob can engage.

(Note that this is like substitutional quantification in logic, which is equivalent to normal quantification over a finite range, here over the finitely many objects in a system, but is not the same over, for example, an uncountable range when the language is countable.)

A combination of these behaviour expressions expresses the behaviour of ob in terms of all the other c-objects $\mathtt{ob}_{\ 1}$ and $\mathtt{ob}_2$. The combination of expression $\mathcal{B}[c_{\ t}.\iota/oid][c.y/c.oid_{\ 1}]$ with expression $\mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 2}]$ may take different forms according to the specific language in which $\mathcal{B}$ is written. In LOTOS, the combinator is '[]' [4], and the combined expression is

$$\mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 1}] \ [] \ \mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 2}]$$

If a logical language is used, then the combination is *logical disjunction*, yielding

$$\mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 1}] \vee \mathcal{B}[c_t.\iota/oid][c.y/c.oid_{\ 2}]$$

This states that ob can engage in $\mathcal{B}$ behaviour with *either* $\mathtt{ob}_{\ 1}$ *or* $\mathtt{ob}_2$. Finally, if the expression language is a finite automaton description language, then the combination operator is the disjoint sum of two automata, but with shared start states (normally only one, but a predicate-action diagram may have more than a single start state [15]).

Although this operation has been expressed with one template and two instantiated objects for simplicity, it generalises directly to multiple class templates and instantiated objects. Whatever its specific form, this syntactic operation of *substitution + combination* yields the behavioural description of object ob at any state of the system. As other objects are created, a similar operation must be performed: say $\mathtt{ob}_{\ 3}$ is newly instantiated with identifier $\mathtt{ob}_{\ 3}.ident$, denoted $c.oid_3$ for simplicity. The expression $\mathcal{B}[c_{\ t}.\iota/oid][c.y/c.oid_{\ 3}]$ must be formed and *combine*d with

[4] but see the comment about [] and choice in Section 3.4.

the other behaviour expressions to yield the behavioural description of object ob in the new environment.

Note that the expressions described here need not actually be formed. It suffices that a procedure exists to obtain a precise description of the behaviour of an object in its run-time environment from the behaviour expression $c_t.\mathcal{B}$ in the class template c_t. We have given such a procedure.

To illustrate the procedure in LOTOS, we introduce part of a process definition which specifies the class template Counter_Teller:

```
process Counter_Teller[t, a](id: Id_Tellers) : noexit :=
  ( t !get_balance !id ?acc_nr: Account_number;
    a !get_balance !acc_nr ?b: Balance;
    t !rtn_balance !id !acc_nr !b;
    exit(id)
   []
     ...
    ) >> accept id: Id_Tellers in Counter_Teller[t, a](id)
  endproc
```

Objects of the class template Counter_Teller invoke services of objects of the class template Account. Let Counter_Teller be the class template c_t, the value id be the object identifier of the current object, while the LOTOS-variable acc_nr: Account_Number corresponds to the individual variable $c.y$.

An expression describing the behaviour of an object instantiated from Counter_Teller, having, say, identifier 125, i.e. the expression corresponding to $\mathcal{B}[Counter\_Teller.\iota/125]$, is

```
t !get_balance !125 ?acc_nr: Account_number; ...
```

Counter teller number 125 can communicate with any object instantiated from the class template Account. Suppose we create precisely two accounts with identifiers 35467 and 35468. The complete behaviour of counter teller 125 when communicating with any of the two accounts, asking for any service, is given by the combination of the services offered by Counter_Teller. We substitute 35467 and 35468 for the expression acc_nr, and combine with the choice operator [] to obtain

$$\mathcal{B}[Counter\_Teller.\iota/125][acc\_nr/35467] \;[]\; \mathcal{B}[Counter\_Teller.\iota/125][acc\_nr/35468]$$

Notice that the behaviour of Counter Teller 125 changes with time, according to the services executed. Suppose we created the two accounts, one subsequently to the other. When communicating with account number 35467, before account 34568 was created, Counter Teller 125's behaviour is described by: $\mathcal{B}[Counter\_Teller.\iota/125][acc\_nr/35467]$. After account 35468 is created, the behaviour expression changes to include in the combination the expression $\mathcal{B}[Counter\_Teller.\iota/125]$ $[acc\_nr/35468]$.

This informal description has illustrated how a description of the behaviour of an object in a specific system may be obtained by purely syntactic means from the behaviour description $\mathcal{B}$ in the class template. The description has been given in a generic way and the particular syntactic operations needed to effect this for the given behavioural specification language LOTOS have been used as illustration. This shows how the behaviour of object ob instantiated from template c_t is determined by the behavioural description $c_t.\mathcal{B}$. A formal definition of ob needs to specify no extra behavioural component beyond that contained in $c_t.\mathcal{B}$.

4.4 *Is_instance*, Classes and Other Concepts

The *is_instance* relation is a binary relation between objects and the class templates of which they are instances. When an object is created, according to the semantics it is a pair of names $ob = \langle identifier, class\_template.name \rangle$. The image of ob under the *named_by* relation [25, page 123] ($named\_by \circ ob$), is thus the pair consisting of the object along with the class template it

was instantiated from. This pair is an element of *is instance*. If one requires this relation for any purpose, it is necessary to require of the object-generator that when it creates an object ob, it also adds (*named by* ∘ ob) to *is Instance* (which is defined to have value ∅ when no objects have yet been created). One may then straightforwardly define the class generated by c_t, $class(c_t) = dom(is\_instance \rhd \{c_t\})$.

An account of the concepts of an OO system will also include a semantics for *message-passing* and for *inheritance*, which require some more detailed attention than the rather simple definition of class and *is instance* above. Since our focus in this paper is on the basic concepts of analysis, not on design or programming, we refer the reader to [18] for a treatment of some of these other concepts.

5 Conclusion

In this paper, we have given a simple denotational semantics for the concepts of object-oriented analysis (OOA), which include *class template*, *service*, *attribute*, *behaviour*, *object generator* and *visibility*. We described the form required of a *behaviour* expression in a class template, so that the behaviour of any object instantiated from that template is precisely specified. We employed the notions of *individual variable* and *substitution* from the syntax of logic to explain in general terms how this behavioural description was transformed. We illustrated this transformation with ROOA templates and behaviour expressions written in LOTOS.

References

[1] J.P. Bahsoun, S. Merz, and C. Servieres. A framework for programming and formalizing concurrent objects. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering: ACM Software Engineering Notes*, 18(5):126-137. ACM Press, December 1993.

[2] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[3] P. Coad and E. Yourdon. *Object Oriented Analysis*. Yourdon Press, Prentice-Hall, 2nd edition, 1991.

[4] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press, Prentice-Hall, 1991.

[5] E. Cusack and M. Lai. Object-oriented Specification in LOTOS and Z or, my Cat Really is Object-Oriented! In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 179–202. Springer-Verlag, 1991.

[6] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and Their Specification. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 40–65. Springer-Verlag, 1993.

[7] H.-D. Ehrich, J.A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 203–228. Springer-Verlag, 1991.

[8] H.-D. Ehrich, A. Sernadas, and C. Sernadas. Objects, Object Types, and Object Identification. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science*, volume 393 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 1989.

[9] R. Gotzhein. *Open Distributed Systems.* Vieweg Verlag, Wiesbaden, 1993.

[10] ISO. Information Processing Systems – Open Systems Interconnection – LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, International Standard 8807. ISO, 1988.

[11] I. Jacobson. *Object-Oriented Software Engineering — A Use Case Driven Approach.* Addison-Wesley, 1992.

[12] C.B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[13] L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, November 1993.

[14] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 1994. To appear.

[15] L. Lamport. TLA in Pictures. January 1994. Preprint.

[16] A. Laorakpong and M. Saeki. Object-Oriented Formal Specification Using VDM. In S. Hishio and A. Yonezawa, editors, *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 529–543. Springer-Verlag, 1993.

[17] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[18] A.M.D. Moreira and R.G. Clark. LOTOS in the Object-Oriented Analysis Process. In *BCS-FACS Workshop on Formal Aspects of Object-Oriented Systems*, Imperial College, London, December 1993. *BCS-FACS (British Computer Society – Formal Aspects of Computing Science).*

[19] A.M.D. Moreira and R.G. Clark. ROOA: Rigorous Object-Oriented Analysis. Technical Report CSM-109, Department of Computing Science and Mathematics, University of Stirling, Scotland, October 1993.

[20] A.M.D. Moreira and R.G. Clark. Combining Object-Oriented Analysis and Formal Description Techniques. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, volume 821 of *Lecture Notes in Computer Science*, pages 344–364. Springer-Verlag, 1994.

[21] J. Reif and S.A. Smolka. The Complexity of Reachability in Distributed Communicating Processes. *Acta Informatica*, 25:333–354, 1988.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall, 1991.

[23] S. Shlaer and S.J. Mellor. An Object-Oriented Approach to Domain Analysis. *ACM Software Engineering Notes*, 14(5):66–77, July 1989.

[24] S. Shlaer and S.J. Mellor. *Object Lifecycles — Modeling the World in States.* Prentice-Hall, 1992.

[25] J.M. Spivey. *The Z Notation.* Prentice-Hall International, 1989.

[26] K.J. Turner, editor. *Using Formal Description Techniques.* John Wiley & Sons, 1993.