OBJECT-BASED CONCURRENCY : A PROCESS CALCULUS ANALYSIS[1]

Elie NAJM
Institut National de Recherche en Informatique et Automatique (INRIA)
Domaine de Voluceau Rocquencourt
BP 105
78153 Le Chesnay Cedex - FRANCE
e-mail : najm@inria.inria.fr

Jean-Bernard STEFANI
Centre National d'Etudes des Télécommunications (CNET)
38-40 rue du Général Leclerc
92131 Issy-les-Moulineaux - FRANCE
e-mail : jbs@tchang.cnet-pab.fr

ABSTRACT

This paper investigates various object-based concepts in a process calculus framework. The principal motivation of this work lies in the need to formally analyze features exhibited by distributed object-based systems and languages. While the dimension of inheritance has been the focus of much attention lately, this paper focuses primarily on object-based features related to concurrency. A simple language is presented, together with its abstract syntax and semantics, that captures basic informal insights about the nature of object-based concurrency. Features analyzed in this paper comprise : object identity, object creation, object communication and object concurrency.

Introduction

Object-basis and object-orientation are now ubiquitous in distributed systems programming and specification. A large number of existing distributed systems and languages for programming distributed systems (for instance ABCL1 [Yonezawa 87], POOL [America 87], ANSA [ANSA 89], Emerald [Black 87], [Raj 88], Argus [Liskov 88], Clouds [Dasgupta 88]) use an object-based approach. The survey [Bal 89] on programming languages for distributed computing systems identifies some twenty different object-based, object-oriented and actor languages. In some research communities, object-based and object-oriented specification is actively considered. For instance [Cusack 89] and [Mayr 88] consider object-oriented extensions to the standardized LOTOS language[2] (Cf [ISO 88]); [Cusack 90] considers issues of typing and inheritance in the context of the LOTOS and the Z languages ; [Duke 90] consider object-oriented extensions to the Z specification language.

Following [Wegner 87], a language is called informally object-*based* if it offers a notion of *object* as a language primitive. An object can be informally defined as a self-contained unit of structure which encapsulates both data and behavior, and which interacts with its environment exclusively through well-defined entry points (messages, methods, operations, features, etc.). A language is called object-*oriented* if it is object-based and if, in addition, it supports (some form of) inheritance. [Wegner 87] informally distinguishes six different dimensions of object-based design : *objects*, *types*, *delegation* (which covers

[1] This work was supported in part by ESPRIT Project n°2267 Integrated Systems Architecture (ISA).

[2] LOTOS combines a CCS-like (Cf [Milner 89]) and CSP-like (Cf [Hoare 1985]) process calculus with the algebraic specification language ACT-ONE (Cf [Ehrig 85]).

inheritance), *abstraction*, *concurrency* and *persistence*. This classification and the nature of its various dimensions is largely subject to discussion, but it is useful here to clarify the scope of this paper: it is only concerned with the *objects* and *concurrency* dimensions[3] . Indeed, while there has been numerous attempts to formalize inheritance and typing features of object-oriented languages, object-based concurrency has received (relatively) less attention. In particular, we are interested to analyze object-based concurrency features in the framework of process calculi such as CCS (Cf [Miner 89]) or CSP (Cf [Hoare 85]).

Relation with other work

The work around the semantics of the object-based language POOL (Cf [America 87] for a presentation of the POOL language) is close to ours. [America 88] investigates object creation and object-based parallelism using operational and denotational models which are built on complete metric spaces and which involve the use of continuations. In [America 88], communication between objects is CSP-like. [Rutten 88] proceeds, using the same approach, to model actual POOL communication constructs which are based on message passing with method invocation. Other works on object-basis and object-orientation which tackle object concurrency comprise [Sernadas 89], [Fiadeiro 90] and [Goguen 90]. These use a category-theoretic approach to semantics. In [Goguen 90] for instance, objects are sheaves, systems are diagrams of sheaves, and the behavior of a system is given by the (categorical) limit of its diagram. In [Fiadeiro 90], behaviors can be specified using a deontic logic of action, and composition of object descriptions is achieved through colimits of (categorical) diagrams.

By contrast to these works, this paper uses standard transition systems and bisimulation as basic semantical tools. In fact, the work with which our approach may end up having the closer relationship is the one concerned with the modelling of dynamic communication structures in process calculi. This work was initiated by Engberg and Nielsen with the introduction of ECCS [Engberg 86], an extension of CCS with label passing. More recently, Milner et al [Milner 89a, 89b, 90] presented their Mobile Processes Calculus which uses, like ECCS, a semantical approach based on transition systems and bisimulations. In contrast to ECCS, the Mobile Processes calculus, and more specifically the π-calculus [Milner 90], aim at capturing the λ-calculus in the process algebra framework.

We are presently investigating the relationship between our formalism (which is called OL1) and the π-calculus. This will be the subject of a forthcoming paper. For the time being, let us note that both formalisms take different approaches. In particular, the π-calculus uses what Milner et al call "late instantiation", while by comparison OL1 uses "early instantiation", as in standard CCS. We consider this to be an advantage. Indeed, in OL1, contrary to the π-calculus, we avoid the following phenomenon, which we would like to call "distributed α-conversion". Let Q, $P = \bar{x}y.P'$ and $R = x(z).R'$ be mobile processes ; then, using the Mobile Processes calculus inference rules we can derive the following internal transition :

$$((y)(P\,|\,Q)\;|\;R\,) - \tau \rightarrow (w)(P'\{w/y\}\,|\,Q'\{w/y\}\,|\,R'\{w/z\}).$$

We can see that in this transition, even though communication occurs only between P and R, Q must somehow participate in the interaction, so as to "agree" on the name, *w*, of the new common internal gate between P, Q, and R.

The paper is organized as follows:

Section 1 discusses various features which are common to numerous object-based languages and which serve as a motivation for the simple language we introduce in section 2. Section 2 describes a simple language, which is called OL1, and which embodies the essential features of object-based concurrency, namely object identity, communication based on object identifiers, and dynamic object creation. We give a formal semantics for OL1 in the form of classical Structured Operational Semantics rules à la Plotkin, and

[3] Although this paper deals only with the objects and concurrency dimensions of object-based languages we expect our framework to be a useful basis for investigating the semantics of concurrent object-*oriented* languages,via the use of delegation which allows to model inheritance.

indicate various properties of the different semantical operators introduced. We provide some examples which illustrate how easily standard object-based concurrency feature are captured with our approach. We introduce a notion of bisimulation which abstracts away from the details of the naming scheme used in OL1. Finally, we conclude with an indication of future directions of research.

1 - Features of object-based concurrency

[Wegner 87] distinguishes several sorts of concurrency in object-based languages. His analysis, however, does not indicate the common features of object-based concurrency. While there is a wide variation between the various languages proposed in the litterature (again see [Bal 89] for an extensive bibliography),the following characteristics can be found in most concurrent object-based systems and languages.

First, concurrent object-based languages possess a number of common supporting features. These are :

- object identity : each object is endowed with a unique *object identifier.* This identifier is generated dynamically at object creation time. Various schemes can be used to ensure the uniqueness of an object identifier (object-id), especially in a distributed system. In the semantics of OL1, we chose one but we show in our definition of *matching bisimulation* how one can abstract from any particular naming scheme. Object identifiers are used as basic types. In particular, as most of these languages are imperative ones with assignment, object identifiers can be used in assignement statements as in :

  x := Exp

where *x* is a variable and *Exp* is an expression which evaluates to an object-identifier. In OL1, we do not have explicit assignment statements ; however, variables can take object identifiers values. Indeed, object identifiers are the only values present in OL1. Object identifiers can be used as references for the construction of arbitrarily complex processes and data structures. This is a general characteristics of object-based languages. For example definitions of dynamic concurrent structures such as trees in object-based concurrent languages see e.g. [America 87] and [Yonezawa 87]. We give examples of this facility in OL1 in section 2.3.

- explicit (and dynamic) object creation : objects are created dynamically through the invocation of explicit creation constructs. Again, various schemes are proposed for object creation in object-based languages (e.g. use of factory objects, "new" constructs in the language). In general, one can find statements such as :

  x := new C

where x is a variable, and C is a class name (i.e. a reference to an object class description). In OL1 , this statement is mirrored by the construct new A(..)<<x>>, where A is an agent name, which plays the same role as a class reference.

Based on these supporting features, the following are almost always found in concurrent object-based languages :

- explicit, pairwise communication between named objects : communication takes place when one object *invokes* the services of another object (through the *invocation* of operations, methods, routines, etc.). This invocation can only take place when the requesting object has obtained the object identifier of its correspondent. Some languages (e.g. ABCL1) offer constructs that ressemble multicasting, but these can be directly simulated by a sequence of pairwise invocations. A communication with a known object can in general be expressed using statements similar to :

  x := z.op(args)

where *z* is a variable that denotes an object identifier (the known object), *op* is the name of an operation, and where *x* is a variable that will be assigned the value returned by the execution of operation *op* with argument *args*, which is invoked on the object referenced by *z.*

- implicit parallelism between objects : objects are implicitly assumed to run concurrently. This is in sharp contrast with structured languages such as CCS and CSP where concurrency is given as an explicit composition means between processes. Equivalently, one can consider that in concurrent object-based languages there exists only one (commutative and associative) composition operator.

The combination of these features in a concurrent object-based language enables the construction of very dynamic process structures. We give in section 2.3 a number of examples of this dynamicity. This again is in contrast with standard CCS or CSP based languages such as LOTOS where the structural operators (parallel composition, disabling, and hiding in the case of LOTOS) are static : process structures are preserved over time. The basic types of change of structure which one can encounter with languages with static structural operators are illustrated in [Milner 89] with what is called there "systems with evolving structure" and "systems with inductive structure".

The semantics of these different constructs and features in object based languages is in general not formalized (with the exception of the POOL language). Our goal then, is to capture and formalize them in a process calculus framework à la CCS, with transition systems and bisimulation semantics.

2 - A simple concurrent object-based language : OL1

We present in this section a simple concurrent object-based language that exhibits the different features mentionned in the preceeding section, namely :

- object-identity with explicit object creation (through a "new" operator) ;
- implicit parallelism[4] between objects with pairwise rendez-vous communication based on object identifiers.

2.1 - Introduction

Object-Expressions

The basic element, the behaviour of which is analysed in OL1, is the object-expression. An object-expression represents a collection of interacting objects. A single OL1 object is the simplest form of object-expression. It is modelled by a triplet $(u, \Theta : B)$ where u is a name - the object-identifier, B is a behaviour-expression representing the current state of u (behaviour-expressions are discussed later), and Θ is the set of names of the objects already created by u. Object-expressions are constructed from objects using the infix, binary, commutative and associative operator $\mid$. Thus, the general form of object-expressions is:

$$(u_1, \Theta_1 : B_1) \mid (u_2, \Theta_2 : B_2) \mid \dots \mid (u_n, \Theta_n : B_n)$$

where $i \neq j \Rightarrow u_i \neq u_j$ (no 2 objects of an object-expression can have the same identifier).
Note: we will use $(u: B)$ to stand for $(u, \varnothing : B)$.

Actions, States and Transitions

An OL1 object, $(u, \Theta : B)$, may change state only as a result of his execution of some action. Using the labelled transition systems paradigm, the execution by $(u, \Theta : B)$ of an action α, can be written:

$$(u, \Theta : B) \xrightarrow{\alpha} (u, \Theta : B') \qquad \text{where } B' \text{ is the next state of } u.$$

[4] In OL1 the parallel operator is not explicit in the sense that it is not part of the syntax of the language. Parallelism, however, is manipulated explicitly since creating a set of objects implies putting them in parallel among themselves and with other existing ones.

The execution of an action by an object may, in some cases, result in the creation of new objects. In these cases, the transition have the following form:

$$(u, \Theta : B) \xrightarrow{\alpha} (u, \Theta' : B') \parallel (u_1, \Theta_1 : B_1) \parallel \ldots \parallel (u_n, \Theta_n : B_n)$$

where, naturally, $\Theta_1 = \Theta_2 = \ldots = \Theta_n = \varnothing$ and $\Theta' = \Theta \cup \{u_1, \ldots, u_n\}$. Thus the above transition can also be written:

$$(u, \Theta : B) \xrightarrow{\alpha} (u, \Theta' : B') \parallel (u_1 : B_1) \parallel \ldots \parallel (u_n : B_n)$$

Note that the creator and the created objects are put in parallel.

Internal actions, invocation actions

An action performed by an object can be either an internal action - denoted by the symbol i, or an invocation action. Invocation actions have the format: (u: v g w) where u is the name of the object performing the action (i.e. the invoking object), v is the name of the object invoked in this action, g is the operation (or gate) which is invoked, and w is a value - an object-name - offered in the invocation. An obvious requirement is that $u \neq v$ (i.e., an object cannot invoke himself).

An example of a transition involving an invocation: $(u, \Theta : B) \xrightarrow{(u: v\, g\, w)} (u, \Theta : B')$.

Interactions - internal to an object-expression

The objects of an object-expression interact among themselves and with their environment. An interaction is a rendez-vous between a pair of complementary invocations. Two invocations (u1 : v1 g1 w1) and (u2 : v2 g2 w2) are complementary iff u1=v2, u2=v1, g1=g2 and w1=w2. The rendez-vous between a pair of complementary invocations results in an invisible action i. Example:

Given the two transition, where S1 and S2 are object-expressions:

$$(u_1, \Theta_1 : B_1) \xrightarrow{(u_1 : u_2\, g\, w)} S1$$
and
$$(u_2, \Theta_2 : B_2) \xrightarrow{(u_2 : u_1\, g\, w)} S2,$$

One can write the resulting transition: $(u_1, \Theta_1 : B_1) \parallel (u_2, \Theta_2 : B_2) \xrightarrow{i} S1 \parallel S2$

Interactions - with the environment of an object-expression

Let $(u_1, \Theta_1 : B_1)$ be an object ready with a firable transition $(u_1, \Theta_1 : B_1) \xrightarrow{\alpha} S1$. Assume that this object is an element of a larger object-expression S, i.e., for some S', $S = (u_1, \Theta_1 : B_1) \parallel S'$. In order for α to be a possible <u>visible</u> action of S, or, in other words, in order for S to perform α, u_1 being the acting object of S performing α, the following condition must hold:

"no object of S' should be invoked in α"

Under this condition, the following transition is derivable from S as a whole:

$$(u_1, \Theta_1 : B_1) \parallel S' \xrightarrow{\alpha} S1 \parallel S'$$

One may say, in this case, that the transition: $(u_1, \Theta_1 : B_1) \xrightarrow{\alpha} S1$, is not restricted or is *immersible* in S. Thus, an object-expression cannot be (externally) invoking one of its own object components.

A concrete example of immersion:

$$(u_1, \Theta_1 : B_1) \xrightarrow{(u_1 : u\, g\, w)} S \ \underline{\text{and}}\ u \neq u_2$$
$$\underline{\text{implies}}$$
$$(u_1, \Theta_1 : B_1) \,\|\, (u_2, \Theta_2 : B_2) \xrightarrow{(u_1 : u\, g\, w)} S \,\|\, (u_2, \Theta_2 : B_2)$$

The combination of immersion and rendez-vous is similar to the combination of restriction/hiding and rendez-vous operators in other process calculi - with the difference that, in OL1, these two types of behaviours are conveyed with the single parallel operator **‖**.

Behaviour-expressions

Behaviour-expressions are elements of a set of terms called $\mathbb{B}$(OL1). The simplest form of behaviour-expression is the constant: **stop**. It represents inaction: object (u, Θ : stop) is the dead object which is no more able to interact with another object.

Action-prefix

Action-prefix is the unary operator of $\mathbb{B}$(OL1) used to describe actions. The general format of action-prefix is: **a; B**, where **a** is an action-denotation and **B** a behaviour-expression. Action-denotations can take various forms: internal action, simple invocations, invocation of unknown objects and invocations with object creation.

Internal action

This is the simplest form of action-denotation. It is noted: i, and the associated action-prefix: **i; B**. The transition:

$$(u, \Theta : i; B) \xrightarrow{i} (u, \Theta : B),$$

is the only possible transition of object u at state **i; B**.

Simple invocations

Simple invocations can take two forms: **g.v !w** and **g.v ?y**, where **g** is a gate - the invocation gate - and **v** is an object-name - the invoked object's name.

In invocation: **g.v !w**, which is called *value offer* invocation, **w** is the value (an object-name) offered in the invocation. Thus, the unique possible transition of object (u, Θ : g.v !w ; B) is:

$$(u, \Theta : g.v\ !w\ ; B) \xrightarrow{(u{:}v\, g\, w)} (u, \Theta : B).$$

In invocation: **g.v ?y**, which is called *value accept* invocation, **?y** indicates that any value (object-name) can be offered: the value offered being stored in the variable y. Note that **?y** is a binding occurence of y.
Let NN be the set of object-names, the possible transitions of object (u, Θ : g.v ?y ; B) take the form:

$$(u, \Theta : g.v\ ?y\ ; B) \xrightarrow{(u{:}v\, g\, w)} (u, \Theta : B[\![w/y]\!]),$$

where $w \in$ NN and $[\![w/y]\!]$ is the substitution in B of y by w.

Note that in actions resulting from **g.v !w** and **g.v ?y**, no reference is made to symbols "?" or "!", i.e., in both cases, values are offered: the single value w in the case of **!w** and all the values of NN in the case of **?y**.

Simple invocations can be used for modelling both pure synchronisation and value-passing:

Pure synchronisation: as presented in a previous section, synchronisation takes place between two objects on their complementary actions. Using action-denotations, this can be illustrated by the following transition:

$$(u_1, \Theta_1\colon g.u_2\,!w\,;\,B_1) \,\|\, (u_2, \Theta_2\colon g.u_1\,!w\,;B_2) \xrightarrow{\;i\;} (u_1, \Theta_1\colon B_1) \,\|\, (u_2, \Theta_2\colon B_2)$$

since $(u_1, \Theta_1 : g.u_2\,!w\,;\,B_1) \xrightarrow{(u_1 : u_2\, g\, w)} (u_1, \Theta_1 : B_1)$,
and $(u_2, \Theta_2 : g.u_1\,!w\,;\,B_2) \xrightarrow{(u_2 : u_1\, g\, w)} (u_2, \Theta_2 : B_2)$.

Note that synchronising objects mutually refer but that they share the same gate and value.

Synchronisation with value passing: synchronisation may also result in a communication between two objects. This is achieved using a couple of invocations: one value offer and one value accept. For instance:

$$(u_1, \Theta_1\colon g.u_2\,!w\,;\,B_1) \,\|\, (u_2, \Theta_2\colon g.u_1\,?y\,;B_2) \xrightarrow{\;i\;} (u_1, \Theta_1\colon B_1) \,\|\, (u_2, \Theta_2\colon B_2[\![w/y]\!]).$$

Invocation of "unknown" objects

In OL1, invoking (an object) and accepting an invocation (from an object) are synonyms. This is due to the symmetric structure of the action where the direction of the value-passing is not recorded. An object with an invocation of the form **g.v ?y** is considered to "offer" to object v any value in NN. Similarly, in OL1, one can be, in one action-denotation, invoking (or accepting the invocation of) any other object. This is achieved by using the ***x** "wild card" notation, where x is any name-variable, to stand for the invoked object. The 2 associated forms of action-prefix are: **g.*x ?y ; B** and **g.*x !w ; B**.

For instance, object $(u, \Theta\colon g.{*}x\,!w\,;B)$ can invoke (or accept the invocation of) any other object v (v must be different from u). Thus, the possible derivations from object $(u, \Theta\colon g.{*}x\,!w\,;B)$ are:

$$(u, \Theta\colon g.{*}x\,!w\,;B) \xrightarrow{(u{:}v\, g\, w)} (u, \Theta\colon B[\![v/x]\!]) \qquad \text{with } v \in NN - \{u\}$$

Note that in the above transitions, u is ready to synchronise with any other (unknown) object having a complementary invocation. Note also that the name of the responding object is stored in x (*x is a binding occurence) and thus becomes known to u and can be used in the subsequent behaviour of u.

Agent Definitions and Instantiations

An agent definition is an equation $A(x_1, \ldots, x_k)\langle x\rangle =_{def} B$, where **A** is an agent-name, **B** is a behaviour-expression - the defining behaviour-expression of **A**, and $x, x_1, \ldots, x_n$ are name-variables - the parameters of **A**. Furthermore, **B** and $x, x_1, \ldots, x_n$ satisfy the condition: $FV(B) \subset \{x, x_1, \ldots, x_n\}$, where $FV(B)$ denotes the set of free variables (as defined later in this text) of **B**.

An agent-instantiation is a behaviour-expression of the form $A(v_1, \ldots, v_k)\langle v\rangle$ where **A** is an agent-name and $v, v_1, \ldots, v_k$ are object-names. The behaviour of an agent-instantiation is obtained from its defining behaviour-expression using the proper replacement of parameters by their values. Example:

Given the agent definition $A(x_1, \ldots, x_k)\langle x\rangle =_{def} B$, then the behaviour of object $(u, \Theta\colon A(v_1, \ldots, v_k)\langle v\rangle)$ is, by definition, the same as $(u, \Theta\colon B[\![v/x, v_1/x_1, \ldots, v_n/x_n]\!])$.

Actions with object creations

Agent definitions can be used, in action denotations, for assigning behaviours to newly created objects. The general syntactical form of action denotations with object creations is: **a newA(z_1, ..., z_k)<u>**, where **a** is any action denotation (which may already contain an object creation), u is an object-name - the identifier of the object being created (the generation of this name is discussed later) and z_1, ..., z_k are either names or name-variables.

The behaviour of: **a newA(z_1, ..., z_k)<v> ; B**, is obtained 'incrementally' from the behaviour of: **a ; B**, in the following manner:
<u>If</u>

$$(u, \Theta : a; B) \xrightarrow{\alpha} (u, \Theta' : B[\![\sigma]\!]) \mid S'$$

with S' an object-expression (containing objects created in a) and σ a (possibly empty) syntactical substitution,

<u>then</u>

$$(u, \Theta : a\ newA(z_1, ..., z_k)<v> ; B) \xrightarrow{\alpha} (u, \Theta' : B[\![\sigma]\!]) \mid (v: A(z_1, ..., z_k)<v> [\![\sigma]\!]) \mid S'$$

Notes:
- the created object v is present only after the completion of the action α.
- the names of new objects are generated before their creation. Thus, since Θ is the set of names already generated by u, the name v assigned to the behaviour A(...)<v>$[\![\sigma]\!]$ belongs to Θ. This implies that $\Theta' = \Theta$. The detailed justification of this fact is given subsequently.

A concrete example of object creation:
<u>Given</u>:

$$(u, \Theta : g.v\ ?y ; B) \xrightarrow{(u:v\,g\,w)} (u, \Theta' : B[\![w/y]\!]),$$

<u>one can infer</u>:

$$(u, \Theta : g.v\ ?y\ newA(y)<u'>; B) \xrightarrow{(u:v\,g\,w)} (u, \Theta' : B[\![w/y]\!]) \mid (u', \emptyset : A(w)<v>)$$

Object Naming

In order to generate different names for different objects, a naming function ν: NN $\times$ $2^{NN} \rightarrow$ NN is used. For u $\in$ NN and $\Theta \in 2^{NN}$, $\nu(u,\Theta)$ is the name generated by object u when Θ is the set of already generated names. ν satisfies the following property: $(u \neq u')$ or $(\Theta \neq \Theta') \Rightarrow \nu(u,\Theta) \neq \nu(u',\Theta')$.

<?x>B is the construct for getting a new name for **x** and binding all free occurences of **x** in **B**. The behaviour of (u, Θ : <?x>B) is equivalent to (u, $\Theta \cup \{\nu(u,\Theta)\}$: B$[\![\nu(u,\Theta)/x]\!]$).

Combining Naming with Creation

Naming and object creation in $\mathbb{B}$(OL1) are not used as presented above, instead they are combined in one syntactical construct:

a newA$_1$(..)<<x$_1$>> ... newA$_k$(..)<<x$_k$>> ; B

which is taken as a shorthand for: **<?x$_1$> ... <?x$_k$> (a newA$_1$(..)<x$_1$> ... newA$_k$(..)<x$_k$> ; B)**.

This combination, together with a static semantics requirement, namely that "**newA(..)<x>** occurences can be bound only by **<?x>** occurences", ensures the uniqueness of object-identifiers.

Choice

This operator, noted: **B + B'**, is similar to the one defined in CCS:
$(u, \Theta : B) \xrightarrow{\alpha} (u, \Theta : B'')$ <u>implies</u> $(u, \Theta : B{+}B') \xrightarrow{\alpha} (u, \Theta : B'')$ <u>and</u> $(u, \Theta : B'{+}B) \xrightarrow{\alpha} (u, \Theta : B'')$

Guards

[z = z'] B : **B** is enabled iff the names z and z' are equal.
[z ≠ z'] B : **B** is enabled iff the names z and z' are different.

Examples

Perhaps the following three examples are the most typical ones of OL1. They illustrate how an object increases his acquaintances by interacting with other objects. These three examples show semantical mechanisms in object systems.

Example 1: In this first example, an object u_2, after interacting with an object u_1, knows about an existing object u_3 (see figure 1a).

$$(u_1 : g.u_2 !u_3 ; B_1) \mid (u_2 : g.u_1 ?y ; B_2) \mid (u_3 : B_3)$$
$$\xrightarrow{i} (u_1 : B_1) \mid (u_2 : B_2[\![u_3/y]\!]) \mid (u_3 : B_3)$$

Example 2: This second example is similar to the first one but involves an object creation: an object u_2, interacts with an object u_1, and thus knows about u_3 ... which is a new object just created by u_1 (see figure 1b).

$$(u_1 : g.u_2 !x' \text{ newA}{<}{<}x'{>}{>}; B_1) \mid (u_2 : g.u_1 ?y; B_2)$$
$$\xrightarrow{i} (u_1, \{u_3\} : B_1[\![u_3/x']\!]) \mid (u_2 : B_2[\![u_3/y]\!]) \mid (u_3 : A{<}u_3{>})$$

Example 3: The third example is a small sophistication of example 2: object u_2 knows about object u_1 but the converse is not true (see figure 1c). However, u_2 invokes u_1 (because u_1, with its g.*x action, is ready to accept the invocation from any object) and an interaction occurs between u_1 and u_2 resulting in:

- u_1 knowing u_2
- the creation of u_3 (by u_1)
- u_2 knows the just created object u_3 and vice versa.

$$(u_1 : g.{*}x !x' \text{ newA}(x){<}{<}x'{>}{>}; B_1) \mid (u_2 : g.u_1 ?y; B_2)$$
$$\xrightarrow{i} (u_1, \{u_3\} : B_1[\![u_2/x, u_3/x']\!]) \mid (u_2 : B_2[\![u_3/y]\!]) \mid (u_3 : A(u_2){<}u_3{>})$$

Conventions :

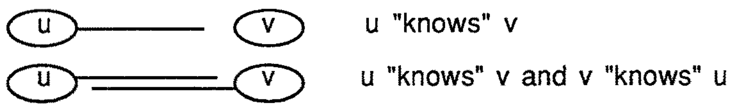


Figure 1a :

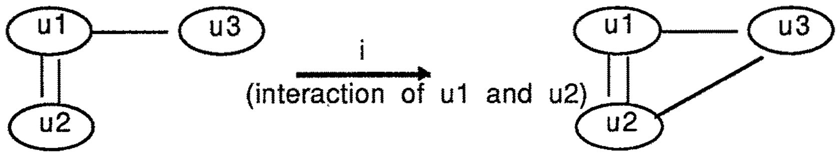


Figure 1b :

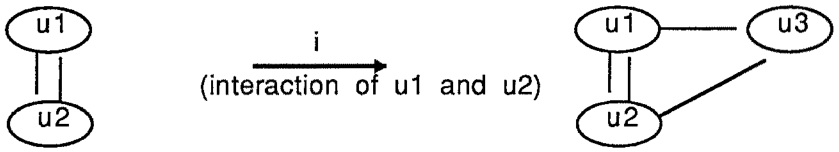
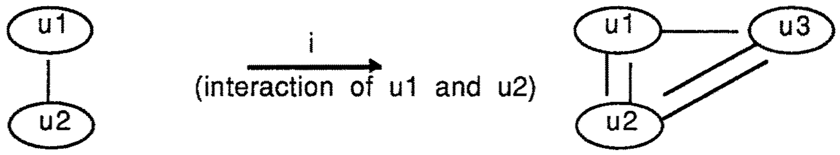


Figure 1c :



2.2 Syntax of OL1

We assume the existence of the following (pairwise disjoint) sets :

- a set NN of *names* (elements from NN will be denoted: u, v, w and will also be called values or object-ids);
- a set VV of *name-variables* or simply variables (elements from VV will be denoted : x, x', y, y', ...) ;
- a set NV of *name-expressions* which is the set union of NN and VV : NV = NN $\cup$ VV (elements from NV will be denoted : z, z', z1,...) ;
- a set G of *gates* also called *operations* (elements from G will be denoted : g, h, g', h', g1, h1,...) ;
- a set $\mathbb{A}$ of *agents* (elements from $\mathbb{A}$ will be denoted : A, A', A1, ...).

Furthermore, for $* \notin$ VV, $*$VV will denote the set obtained by prefixing the elements of VV with the symbol "$*$". Thus, $*VV =_{def} \{*x \mid x \in VV\}$. Finally, Let $NV^+ =_{def} NV \cup *VV$ (elements from NV^+ will be denoted: U, V, W).

Action-denotations is the set, with generic element a, generated by the following grammar:

```
p    :=    !z | ?x
e    :=    i  | g.U p
c    :=    new  A(z1, ..., zk)<<x>>
a    :=    e | ac
```

The set of behaviour-expressions of OL1, denoted by $\mathbb{B}$(OL1) is recursively defined by the following context-free grammar:

B := stop | a ; B | B + B | [z = z'] B | [z ≠ z'] B | A(z_1, ..., z_k)<z>

We further assume a family of defining equations for agents : A(x_1, ..., x_k)<x> $=_{def}$ B, with *FV*(B) ⊂ {x, x_1, ..., x_k} where *FV*(B) denotes the set of free variables (as defined later) of B.

Scopes, Binding and Bound Occurences

Our static (and operational) semantics analysis of OL1 will be based on $\mathbb{T}$(OL1) - an expanded version of $\mathbb{B}$(OL1) where the "a newA_1(..)<<x_1>> ... newA_n(..)<<x_n>>; B" occurences, combining naming and creation of objects, have been replaced by their equivalent decomposition: "<?x_1> ... <?x_n> (a newA_1(..)<x_1> ... newA_n(..)<x_n>; B)".

Thus $\mathbb{T}$(OL1) is defined by: B := stop |<?x> B |a' ; B | B + B | [z=z']B | [z≠z']B | A(z_1, ..., z_k)<z>

where c' := new A(z_1, ..., z_k)<z>

a' := e | a' c'

Scopes, binding and free occurences of variables are treated in $\mathbb{T}$(OL1) in the usual way. Binding occurences are occurences having one of the 3 forms ∗x, ?x or<?x>. If T is a syntactical element, we will use *BD*(T) to denote the set of binding occurences of T. All other forms of occurences are said to be place-marking. A scope is associated to each binding occurence of a variable. The scopes of occurences of the form ?x and ∗x are shown in the following examples, where the underlined text represents the scope, and C a (possibly empty) list of object creations:

- scope of ?x :	g.U	?x	$\underline{C\ :\ B}$,
- scope of ∗x :	g.∗x	p	$\underline{C\ ;\ B}$.

The scopes of any two binding occurences of two variables having the same name x should be such that one is strictly included in the other. As usual, a place-marking occurence of a variable x will be bound to the binding occurence of x having the smallest scope. A place-marking occurence of a variable x is said to be free if it is not contained in the scope of a binding occurence of x. Using *FV*(B) to denote the set of free variable occurences in B, the table below, where C denotes a (possibly empty) list of object-creations, gives a structural definition of *FV*.

B	*FV*(B)
stop	∅
A(z1, ..., zn)<z>	{z1, ..., zn, z} ∩ VV
new A(z1, ..., zn)<z>	{z1, ..., zn, z} ∩ VV
i C ; B'	*FV*(C) ∪ *FV*(B')
g.U p C ; B'	((*FV*(B') ∪ *FV*(C)) - *BD*(U) - *BD*(p)) ∪ *FV*(U) ∪ *FV*(p)
<?x> B'	*FV*(B') - { x }
B1 + B2	*FV*(B1) ∪ *FV*(B2)
[z ≠ z'] B'	*FV*(B') ∪ ({z, z'} ∩VV)
[z = z'] B'	*FV*(B') ∪ ({z, z'} ∩VV)

structural definition of *FV*.

A static semantics rule: In order to ensure the uniqueness of object-identifiers, the following 2 static semantics requirements have to hold:

- occurences of the form " new A(..)<x> " can only be bound by occurences of the form " <?x> ",
- an occurence of the form " <?x> " can bind at most one occurence of the form " new A(..)<x> ".

Shorthand notations: we will use shorthand notations for 𝔹(OL1) actions as shown in the following examples where C is a (possibly empty) list of object creations:

- g C ; B	: is a shorthand for g.∗x ?y C ; B, with x and y not free in B and C;
- g p C ; B	: is a shorthand for g.∗x p C ; B, with x not free in B and C;
- g.U C ; B	: is a shorthand for g.U ?y C ; B, with y not free in B and C.

2.3 - Semantics of OL1

The semantical domain of our language OL1 is a set of labelled transition systems denoted by $\mathcal{LTS}$(OL1). The specific forms of the states and labels of transition systems in $\mathcal{LTS}$(OL1) are discussed hereafter. The mapping of OL1 constructs onto transition systems in $\mathcal{LTS}$(OL1) will be defined using the Structural Operational Semantics paradigm, i.e., a set of derivation rules à la Plotkin.

The Labels of Transition Systems in $\mathcal{LTS}$(OL1)

We define the set ACT(OL1) of labels of $\mathcal{LTS}$(OL1) as follows (elements from ACT(OL1) will be denoted by α, $\alpha 1$, ...):

$$\mathrm{ACT(OL1)} = \{ i \} \cup \{ (u : v\ g\ w) \mid g \in G \text{ and } u, v, w \in NN \}$$

The States of Transition Systems in $\mathcal{LTS}$(OL1)

The set $\mathbb{S}$(OL1) of states of transition systems in $\mathcal{LTS}$(OL1) is defined by the following grammar:

$$S := (u, \Theta : B) \mid S \,\|\, S \quad \text{where } B \in \text{closed\_terms}(\mathbb{B}(\mathrm{OL1})), \text{ i.e., } FV(B) = \emptyset, \text{ and}$$
$$\Theta \text{ is a set of object-names.}$$

We will use the notation (u : B) to stand for (u , ∅ : B).

Elements of $\mathbb{S}$(OL1) will be called object-expressions and will be denoted by S, S', S1, ...

The meaning of an expression B of 𝔹(OL1) is obtained by borrowing a name u from NN: the semantics of B is the labelled transition system rooted at (u : B). The name u can be then abstracted in a second level of semantics.

The Derivation System of $\mathcal{LTS}$(OL1)

Some preliminary definitions are needed prior to the introduction of our derivation rules:
- we define the function REQ : (ACT(OL1) - { i }) → NN by : REQ(u:v g w) = v;
- for S ∈ $\mathbb{S}$(OL1), OBJ(S) denotes the set { u | ∃ B, ∃ Θ such that (u, Θ : B) occurs in S };
- L ⟦d_1/f_1, ... d_n/f_n⟧ denotes the expression obtained from the syntactical element L by replacing in L the free occurences of f_i by d_i;

- a function called *dom* is defined on some of the syntactical elements of OL1 as follows:
  $dom(?x) = dom(*x) = NN$, $dom(u) = \{ u \}$
- furthermore we will use u/*x and u/?x to denote the substitution of x by u, and u/u to denote the identity function.

In order to generate different names for different objects, a naming function $v: NN \times 2^{NN} \rightarrow NN$ is used. For $u \in NN$ and $\Theta \in 2^{NN}$, $v(u,\Theta)$ is the name generated by object u when Θ is the set of already generated names. v satisfies the following property: $(u \neq u')$ or $(\Theta \neq \Theta') \Rightarrow v(u,\Theta) \neq v(u',\Theta')$.

We can now define the different semantical rules for OL1. In the following rules, a transition will be denoted: $S \text{ — } \alpha \rightarrow S'$, where α is an element of ACT(OL1). S and S' are, in general, elements from $\mathbb{S}$(OL1): rule (Dec) decomposes naming and object-creation and as a consequence generates the 2 syntactical constructs "<?x>B" and "a new A(..)<v> ; B" which are not present in $\mathbb{B}$(OL1). However they are "absorbed" in rules (Nam) and (Act4).

Some transtions will be denoted $S \text{ — } \alpha/\sigma \rightarrow S'$ where σ is a syntactical substitution. σ is not really part of the visible label of the transition and is only present in the rules implying action-prefix and object-creation. It is only an accessory notation used to simplify the derivation rules of object-creation. In fact, one extra "technical" derivation rule (Act 4 bis) is included in the set of rules, with the purpose of removing the $/\sigma$ construct from the actions.

Internal action

$$\text{(Act 0)} \quad \frac{}{(u, \Theta : i; B) \text{ — } i \rightarrow (u, \Theta : B)}$$

an offering invocation

$$\text{(Act 1)} \quad \frac{v \in dom(V) - \{u\}}{(u, \Theta : g.V\,!w\,; B) \text{ — } (u{:}v\ g\,w)/\sigma \rightarrow (u, \Theta : B[\![\sigma]\!])} \quad \text{where } \sigma = v/V$$

a receiving invocation

$$\text{(Act 3)} \quad \frac{v \in dom(V) - \{u\} \qquad w \in NN}{(u, \Theta : g.V\,?x\,; B) \text{ — } (u{:}v\ g\,w)/\sigma \rightarrow (u, \Theta : B[\![\sigma]\!])} \quad \text{where } \sigma = v/V,\ w/x$$

decomposing naming and creation

$$\text{(Dec)} \quad \frac{(u, \Theta : <?x_1> \ldots <?x_n> (a\,newA_1(..)<x_1> \ldots\ newA_n(..)<x_n>; B)) \quad \text{— } \alpha \rightarrow S}{(u, \Theta :\ a\,newA_1(..)<<x_1>> \ldots\ newA_n(..)<<x_n>>; B) \quad \text{— } \alpha \rightarrow S}$$

New-Name

$$\text{(Nam)} \quad \frac{(u, \Theta \cup \{v(u, \Theta)\} : B[\![v(u, \Theta)/u]\!]) \text{ — } \alpha \rightarrow S}{(u, \Theta : <?x>B) \text{ — } \alpha \rightarrow S}$$

object creation

$$\text{(Act 4)} \quad \frac{(u, \Theta : a\,; B) \quad \text{— } \alpha/\sigma \rightarrow \quad S}{(u, \Theta : a\ \ new\,A(\ldots)<v>; B\,) \quad \text{— } \alpha/\sigma \rightarrow \quad (v :\ A(\ldots)<v>[\![\sigma]\!]\,) \,|\, S}$$

removal of σ

$$(\text{Act 4 bis})\quad \frac{(u, \Theta : B) \;\; \xrightarrow{\alpha/\sigma} \;\; S}{(u, \Theta : B) \;\; \xrightarrow{\alpha} \;\; S}$$

choice

$$(\text{Cho 1})\quad \frac{(u, \Theta : B1) \xrightarrow{\alpha} S}{(u, \Theta : B1 + B2) \xrightarrow{\alpha} S} \qquad (\text{Cho 2})\quad \frac{(u, \Theta : B2) \xrightarrow{\alpha} S}{(u: B1 + B2\,) \xrightarrow{\alpha} S}$$

Guards

$$(\text{Test 1})\quad \frac{(u, \Theta : B) \xrightarrow{\alpha} S}{(u, \Theta : [v = v]B\,) \xrightarrow{\alpha} S} \qquad (\text{Test 2})\quad \frac{(u, \Theta : B) \xrightarrow{\alpha} S \qquad v \neq w}{(u, \Theta : [v \neq w]B\,) \xrightarrow{\alpha} S}$$

Recursion

$$(\text{Rec})\quad \frac{(u, \Theta : B[\![..w_i/x_i.., v/x]\!]\,) \xrightarrow{\alpha} S}{(u, \Theta : A(..w_i..)<v>\,) \xrightarrow{\alpha} S} \qquad \text{where } A(..x_i..)<x> =_{def} B$$

Immersion left

$$(\text{PAR1})\quad \frac{S1 \xrightarrow{\alpha} S1' \qquad OBJ(S1) \cap OBJ(S2) = \emptyset \qquad \alpha = i \text{ or } (\alpha \neq i \text{ and } REQ(\alpha) \notin OBJ(S2)\,)}{S1 \mathbin{\|} S2 \;\; \xrightarrow{\alpha} \;\; S1' \mathbin{\|} S2}$$

Immersion right

$$(\text{PAR1'})\quad \frac{S2 \xrightarrow{\alpha} S2' \qquad OBJ(S1) \cap OBJ(S2) = \emptyset \qquad \alpha = i \text{ or } (\alpha \neq i \text{ and } REQ(\alpha) \notin OBJ(S1)\,)}{S1 \mathbin{\|} S2 \;\; \xrightarrow{\alpha} \;\; S1 \mathbin{\|} S2'}$$

Rendez-Vous

$$(\text{PAR2})\quad \frac{S1 \xrightarrow{(u:v\,g\,w)} S1' \qquad S2 \xrightarrow{(v:u\,g\,w)} S2' \qquad OBJ(S1) \cap OBJ(S2) = \emptyset}{S1 \mathbin{\|} S2 \;\; \xrightarrow{i} \;\; S1' \mathbin{\|} S2'}$$

A remark on the rule for object creation (Act 4) : the rule (Act 4) which is given above indicates that an object's creation consumes an action , i.e. the creation is done as a side effect of an action either on i or on some operation g. One can also imagine a syntactical construct for object-creation which does not consume any action. The syntax for this construct would be: new A(...)<v>; B. The associated derivation rule is readily written as :

$$(\text{New'})\quad \frac{(u, \Theta : B) \mathbin{\|} (v: A(...)<v>\,) \xrightarrow{\alpha} S}{(u : \text{new } A(...)<v>\, ; B\,) \xrightarrow{\alpha} S}$$

We need to prove first that these rules are well-formed. Indeed, we have required that if B is a term which appears in an object-expression (u, Θ : B), then: B $\in$ closed_terms($\mathbb{B}$(OL1)). The following proposition makes sure our rules are well-formed[5].

__Proposition 1__:

if $S = (v_1, \Theta_1 : B_1) \mid \ldots \mid (v_n, \Theta_n : B_n)$ where $\forall$ i, $1 \le i \le n$, $B_i \in$ closed_terms($\mathbb{B}$(OL1)), and $S \xrightarrow{\alpha} S'$,

then $S' = (u_1, \Theta'_1 : B'_1) \mid \ldots \mid (u_k, \Theta'_k : B'_k)$ where, $\forall$ j, $1 \le j \le k$, $B'_j \in$ closed_terms($\mathbb{B}$(OL1)).

__Proof__: By induction on the length of the inferences which ensure $S \xrightarrow{\alpha} S'$.

We need to prove that our identification scheme for objects is correct (i.e. ensures the uniqueness of generated object-IDs). The proof of the unique identification scheme is in fact the conjunction of the following two propositions :

__Proposition 2__: for all B $\in$ closed_terms($\mathbb{B}$(OL1)), all $\Theta \in 2^{NN}$ and all u $\in$ NN, and for all S $\in \mathbb{S}$(OL1) such that (u, Θ : B) $\rightarrow^*$ S, we have $S = (v_1, \Theta_1 : B_1) \mid \ldots \mid (v_n, \Theta_n : B_n)$ where, for all i,j with $1 \le i,j \le n$, if $i \ne j$ then $v_i \ne v_j$.

__Proof__: By induction on the length of the derivation (u : B) $\rightarrow^*$ S.

__Proposition 3__: If S $\in \mathbb{S}$(OL1) with $S = (u_1, \Theta_1 : B_1) \mid \ldots \mid (u_n, \Theta_n : B_n)$ where for all i, j with $1 \le i,j \le n$, $i \ne j \Rightarrow u_i \ne u_j$, then for all S' such that S->*S' we have S' is in the form $(v_1, \Theta'_1 : B'_1) \mid \ldots \mid (v_k, \Theta'_k : B'_k)$ where, for all i,j, $1 \le i,j \le k$, $i \ne j \Rightarrow v_i \ne v_j$.

__Proof__: By induction on the length of the derivation $S \rightarrow^* S'$.

Finally, we need now to show that our (implicit) parallel operator, $\mid$, possesses the elementary properties we expect for implicit parallel composition of objects: commutativity and associativity. Indeed, these two properties ensure that we can forget the parallel structure of object composition.

__Proposition 4__: $\forall$ S1, S2, S3 $\in \mathbb{S}$(OL1) we have :

(a) $S1 \mid S2 \sim S2 \mid S1$

(b) $(S1 \mid S2) \mid S3 \sim S1 \mid (S2 \mid S3)$

where ~ denotes the standard relation of strong bisimulation between labelled transition systems.

__Proof__: Standard, by devising the right relations.

[5] In the proposition that follows, an object-expression $S = (v_1, \Theta_1 : B_1) \mid \ldots \mid (v_n, \Theta_n : B_n)$ should be interpreted with arbitrary parentheses in place. We prove in proposition 4 that indeed we can forget these parentheses .

2.4 - Examples

The three following examples are written directly in OL1: we do not show the underlying object systems.

Example 4 (Factory object): a *Factory* object for a class B is an object that possesses a *create* operation. On invocation of this operation, a *Factory* object returns to the invoker of the *create* operation the object-id of a newly created object which has behavior B.

```
• Factory = create.*x   new B<<y>> ; r.x ! y ; Factory        /* Factory has gates create and r */
```

A client of a *Factory* object can obtain the object-id of the newly created object on the *r* gate.

Example 5 (FIFO Queue): we define in this example a FIFO Queue. Figure 2 shows a typical resulting object system. We need to define first *Cell* objects which will act as units of storage (they will hold the object-ids of the objects which are "put" in the queue).

```
• Cell(right, left,control, c) =   tap.control ! c ; Cell(right, left,control, c)
                                  + chr.control ? y ; Cell(y, left, control,c)
                                  + chl.control ? y ; Cell(right, y, control, c)
                                  + giver.control ! right ; Cell(right, left, control,c)
                                  + givel.control ! left ; Cell(right, left, control,c)
```

/* Cell has gates tap, chr, chl, giver, givel */

A *Cell* object merely stores an object-id *c*, and maintains pointers towards its *right* and *left* neighbors. By invoking the *tap* operation, the *control* object (and only it) can retrieve the stored object-id. The *control* object has full control over a *Cell* object : it can change its *right* and *left* neighbors (using operations *chr* and *chl* respectively), or it can ask for them (*giver* and *givel* operations respectively).

We can now define FIFO Queue objects. An empty FIFO Queue is represented with our definition by a FifoQ object with parameters (self, self)<self> :

```
• FifoQ(last, first) <self> =
        out.*x ;  (
                     [first ≠ self] tap.first ? y ; ok.x ! y ; givel.first ? y ;
                                                   (
                                                    ( [y ≠ self] chr.y ! self ; FifoQ(last, y) <self> )
                                                   + ( [y = self] FifoQ(last, y) <self> )
                                                   )
                     + [first = self] nok.x ; FifoQ(last, first) <self>)
                   )
        + in ? c new Cell(last, self, self, c)<<x>> ; (
                                                     [last ≠ self] chl.last ! x ; FifoQ(x, first) <self>
                                                     + [last = self] FifoQ(x, x) <self> )
```

/* FifoQ has gates out, in, ok, nok */

When the *in* operation is invoked, a new *Cell* object is created, that holds the reference of the object which is put in the queue, and this newly created *Cell* object becomes the last element of the queue (i.e. its object-id is stored in the *last* parameter of the *FifoQ* object). When an *out* operation is invoked, if the queue is not empty, then the first object in the queue is returned to the invoker on the *ok* gate. If the queue is empty (*first = self* condition) then the *nok* gate is used.

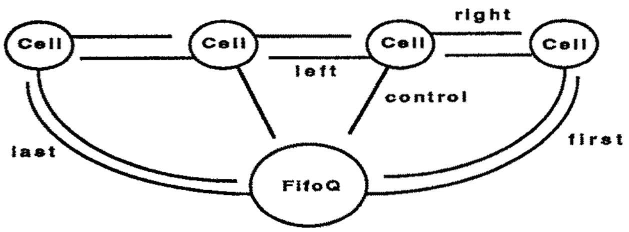


Figure 2

Example 6 (Ring) : we define in this example a ring, i.e. a circular list of objects that can be reconfigured by insertion of objects into, and deletion of objects from the ring. A *Ring* object system is shown on figure 3. One can think of a *Ring* object as some kind of cursor that moves around a circular list of *Cell* objects. An empty ring is represented with our definition by a Ring object with parameters (self, self, self). We use the notion of *Cell* that was given above.

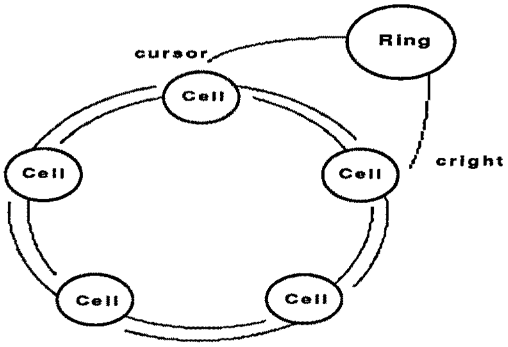


Figure 3

```
• Ring(cursor, cright)<self> =
        [cursor = self] insert ? y  new Cell(x, x, self, y)<<x>> ; Ring(x, x)<self>
        + [cursor ≠ self] insert ? y  new Cell(cright, cursor, self, y)<<x>> ; chr.cursor ! x ; chl.cright ! x ;
                                                                    Ring(cursor, x)<self>
        + [cursor ≠ self] movel ; givel.cursor ? y ; Ring(y, cursor)<self>
        + [cursor ≠ self] mover ; giver.cright ? y ; Ring(cright, y)<self>)
        + [cursor ≠ self] content.*x ; tap.cursor ? y ; ok.x ! y ; Ring(cursor, cright)<self>
        + [cursor = self] content.*x ; nok.x ; Ring(cursor, cright)<self>
        + [cursor ≠ self] delete ; givel.cursor ? y ;
                ( [y = cursor] Ring(self, self)<self>
                 + [y ≠ cursor] chr.y ! cright ; chl.cright ! y ; Ring(y, cright)<self> )
```

/* Ring has gates insert, delete, movel, mover, content, ok, nok */

Note that one cannot *remove* an object from the empty ring. If there is only one remaining object in the ring (condition *left* neighbor of *cursor* is *cursor*) then the *remove* operation turns the ring into an empty one. The *movel* operation just moves the *cursor* to the left. The *mover* operation moves the *cursor* to the right. At any time, *cright* designates the *right* neighbor of the *cursor*.

2.5 - Naming Abstractions

The strong and weak observation equivalences, introduced by [Milner 89], provide a sound semantical framework for analysing systems based on the labelled transition systems model.

The specific labelled transition systems associated to OL1 object-expressions, record in their labels the names of the interacting objects. These names are generated according to the naming scheme chosen for OL1, but any other naming scheme which would preserve the unique naming property of names would do. It is therefore interesting to analyse the meaning of OL1 object-expressions focusing on the role that the involved objects play, while abstracting from the actual names used to identify them and abstracting from the actual naming scheme used. For that purpose, strong and weak observation equivalences may be combined with abstractions (Cf [Boudol 85]).

Definitions and Notations

In an action a=(u:v g w), u is called the *agent* of a.

Let S be an object-expression, S = (u1, $\odot_1$: B1) ‖ ... ‖ (un, $\odot_n$: Bn), then:

- TR(S) denotes the labelled transition system associated to S,
- Names(S) denotes the set of names of objects occuring in the labels of TR(S),
- Initials(S) $=_{def}$ {u1,...,un},
- Agents(S) denotes the set of object-names which are agents in some action of TR(S),
- NewAgents(S) $=_{def}$ Agents(S) \ Initials(S).

An *object- mapping* m (or mapping for short) on NN is a binary relation on NN such that the restriction of m to its domain and codomain is a bijection, i.e., if u1 m u2 and v1 m v2 then u1=v1 $\Leftrightarrow$ u2=v2.

Let $\mathcal{M}$ denote the set of object-mappings. $\mathcal{M}$ is endowed with the natural set inclusion partial order which we will denote by $\leqslant$ and which is defined by: m $\leqslant$ m' $\Leftrightarrow$ (u m v $\Rightarrow$ u m' v).

Each object-mapping m of $\mathcal{M}$ will be considered also as establishing a relation, denoted with the same name m, on ACT(OL1)xACT(OL1)), defined in the following way:
a1 m a2 iff either a1 = a2 = i or a1=(u1:u2 g u3) and a2=(v1:v2 g' v3) where g=g' and for j=1,2,3, uj m vj

Pre-Matching and Matching Bisimulations

Definition: A family $\mathcal{R}$ of relations on $\mathcal{S}$(OL1)X $\mathcal{S}$(OL1) which is indexed by a subset $\mathcal{M}'$ of $\mathcal{M}$, i.e., $\mathcal{R}$={R_m | m $\in$ $\mathcal{M}'$}, is a *monotonic family* iff: $\forall$ m, m' $\in \mathcal{M}'$, if m $\leqslant$ m', then $R_{m'} \subset R_m$.

Definition (Pre-Matching Bisimulation $\approx$): Two object-expressions S1 and S2 are said to be *pre-matching bisimular* (notation : S1 $\approx$ S2) iff there exists a monotonic family $\mathcal{R}$ ={R_m | m $\in$ $\mathcal{M}'$}, and $m_0 \in \mathcal{M}'$ such that:

- S1 R_{m0} S2
- if s1 R_m s2 then $\forall$ a1, r1 where s1 $\xrightarrow{a1}$ r1
  $\exists$ a2, r2, m' such that: m $\leqslant$ m' , s2 $\xrightarrow{a2}$ r2, a1 m' a2, r1 $R_{m'}$ r2,
  and
  $\forall$ a2, r2 where s2 $\xrightarrow{a2}$ r2
  $\exists$ a1, r1, m' such that: m $\leqslant$ m' , s1 $\xrightarrow{a1}$ r1, a1 m' a2, r1 $R_{m'}$ r2.

<u>Proposition 5</u>: The pre-matching bisimulation is an equivalence relation.

<u>Proof</u>: straightforward.

When established between two object-expressions, the pre-matching bisimulation implies that each object in one of the two expressions plays a role which is matched by an object of the second expression. Two pre-matching bisimilar object-expressions may differ in the number and the names of the involved objects, however. As an example, consider:

S1 = (u, {u1} : g.w ; h.u1 ; stop) ∥ (u1 : h.u ; stop) and S2 = (v : g.w ; i ; stop).
S1 and S2 are pre-matching bisimilar but S2 contains only one object, while S1 contains two objects.

Note that the matching of objects is not defined globally between the two object-expressions but gradually and incrementally along with the derivation of actions.

Pre-matching bisimulation is not a congruence since the names of the objects are, to a great degree, relevant for the establishment of communications. Thus, two pre-matching bisimular object-expressions are not always interchangeable in the context of the ∥ operator. As an obvious example, consider: S1 = (u:g.v ! v' ; stop) is pre-matching bisimlar to S2 = (w: g.v ! v' ; stop). However, taking S3 = (v: g.u ! v' ; stop), we have: S1∥S3 $\xrightarrow{i}$ (u: stop)∥ (v: stop), while no transition is possible from S2∥S3.

In fact, not all contexts are worth considering as testbeds for interchangeability of object-expressions. Take for example the object-expression (S∥S1), where we want to consider (S∥ .) as a context for S1. Consider now the case where S "guesses" the names of the objects that are going to be created by S1 before these are created. As a concrete example, let us take:

S = (v: g.u;B) and S1 = (w: i new A[g](v) <<x>>; B1) where A[g](v)<<x>> = g.v ; stop and $\nu(w, \emptyset) = u$
We have the sequence of derivations:

(S ∥S1) $\xrightarrow{i}$ (w, {u}: B1)∥ (u: g.v;stop)∥ (v: g.u;B) $\xrightarrow{i}$ (w: B1)∥ (u: stop)∥ (v: B)
where the second derivation is made possible only because S knows beforehand that S1 is going to create object u.

Good design and style implies that objects acquire information only by interacting with other objects and not by guessing their internal mechanisms for naming created objects. The object-expressions which we will consider in the sequel are *independently formed object-expressions*, i.e., object-expressions where no object has any beforehand information about the names of objects that will be created by other objects of the expression. An object-expression S = (u1, $\ominus_1$: B1)∥ ... ∥ (un, $\ominus_n$: Bn) is said to be *independently formed* iff the following condition is satisfied:

$$\forall\ i,j \text{ with } i \neq j,\ \text{names}(Bi) \cap \nu^*(uj, \ominus_j) = \emptyset$$

where names(B) is the set of constant names (elements of NN) occuring in B, and where $\nu^*(u, \ominus)$ is the set of all names generated by repeated applications of ν, starting with the initial couple $(u, \ominus)$. It can be defined thus:

$\nu^*(u, \ominus) = \bigcup_{j \in \mathbb{N}^+} \nu_j(u, \ominus)$ with ($\nu^*(u, \ominus)$ is the set of all offsprings of u)

- $\nu_j(u,\ominus) = \bigcup_{v \in \nu_{j-1}(u,\ominus)} \nu_{j-1}(v, \emptyset)$ ($\nu_j(u,\ominus)$ is the set of offspringsof u to the jth generation)

- $\nu_1(u, \ominus) = \bigcup_{i \in \mathbb{N}^+} P_i(u, \ominus)$, ($\nu_1(u, \ominus)$ is the set of all sons of u)

- $P_i(u, \ominus) = P_{i-1}(u, \ominus) \cup \{\nu(u, P_{i-1}(u, \ominus))\}$, (Pi is the set of the first i sons of u after $\ominus$)

- $P_0(u, \ominus) = \ominus$.

We can note that this condition is trivially satisfied if we consider only object-expressions that derive from initial object-expressions of the form (u1: A1(...)<u1>) ❙ ... ❙ (un: An(...)<un>) where, for all i, names(Ai(...)<ui>) $\subset$ {u1, ..., un} and where all associated agent-definitions have no constant names.

Definition (matching bisimulation $\approx$) : Two object-expressions S1 and S2 are said to be matching bisimilar (notation: S1 $\approx$ S2) iff

(a) there exists a monotonic family $\mathcal{R}=\{R_m \mid m \in \mathcal{M}'\}$ and $m_0 \in \mathcal{M}'$ such that S1 $\approx$ S2, and

(b) $\mathcal{M}'$ satisfies the following stability condition :

$\forall$ m $\in \mathcal{M}'$, $\forall$ u, v $\in$ NN if u m v then:

- u $\in$ NewAgents(S1) $\Leftrightarrow$ v $\in$ NewAgents(S2)
- u $\notin$ NewAgents(S1) $\Rightarrow$ v = u

Intuitively, two matching bisimilar object expressions possess the same set of initial objects and the same set of created objects. With the following definition, one can extend the matching bisimulation relation to $\mathbb{B}$(OL1) terms :

Definition (matching bisimulation $\approx$ on $\mathbb{B}$(OL1)) : If B1, B2 $\in \mathbb{B}$(OL1) and *FV*(B1) = *FV*(B2) = {x1,..., xn}, B1 and B2 are said to be matching bisimilar (notation: B1 $\approx$ B2) iff

$\forall$ u, v1, ..., vn $\in$ NN (u:B1⟦v1/x1,...,vn/xn⟧) $\approx$ (u:B2⟦v1/x1,...,vn/xn⟧)

We can now prove that the matching bisimulation is a congruence on $\mathbb{B}$(OL1). This is a direct consequence of :

Proposition 6: If S' = S❙S1 and S" = S❙S2 are independently formed object-expressions and S1 $\approx$ S2, then S' $\approx$ S".

Proposition 7: If B1, B2 $\in \mathbb{B}$(OL1) and B1 $\approx$ B2 then:

(i) $\forall$ a	a ; B1 $\approx$ a ; B2
(ii) $\forall$ B	B + B1 $\approx$ B + B2 and B1 + B $\approx$ B2 + B
(iii) $\forall$ a, B	a new A1(...)<<x>> ; B $\approx$ a new A2(...)<<x>> ; B with A1(...)<x> $=_{def}$ B1 and A2(...)<x> $=_{def}$ B2
(iv) $\forall$ a, B	a new A(...)<<x>> ; B1 $\approx$ a newA(...)<<x>> ; B2 with A(...)<x> $=_{def}$ B

Proof: The proofs of Prop 6 and Prop 7 are not included in this paper for the sake of brevity. They are mainly based on the construction of the proper monotonic family of relations which establishes the matching bisimulation in the different cases.

3 - Conclusion

We have presented in this paper a simple language, OL1, which exhibits in a primitive form what we think are the essential features of object-based concurrency : pairwise communication between uniquely named objects, implicit parallelism between objects, dynamic object creation and object systems reconfiguration. At the same time, we have captured also a basic characteristic of object-based languages: the possibility to build arbitrary data structures (in our case, also process structures) out of the manipulation of object identifiers.

Although OL1 has a process-calculus flavor, its essential operators (e.g. ▮) are "hidden" in the semantics of the language. If we were to devise a true "object-calculus", we would need to manipulate these operators directly at the syntactical level. We are presently investigating such a possibility. Along this line of research is the problem of relating OL1 to other existing process calculi such as CCS. In fact, one can see intuitively that a translation of OL1 into CCS is due to be difficult. As we mentioned in section 1, OL1 shares with truly object-based languages the possibility to build very dynamic process structures, whereas in CCS such a facility needs to be simulated explicitly, using non trivial constructions : as mentioned previously, CCS process structures essentially fall into two categories "inductive structures" and "evolving structures" [Milner 89]. This remark has led us very recently to consider instead a mapping of OL1 onto Milner's π-calculus, or calculus of Mobile Processes (see references [Milner 89a], [Milner 89b] and [Milner 90]). Compared to the π-calculus, OL1 does not use the problematic distributed α-conversion which is at the core of the π-calculus. Nevertheless, it turns out that OL1 (or at least a large subset of it) is directly translatable, in some precise sense, into the π-calculus. Conversely, we have been able to devise a second language, OL2, quite similar to OL1 but exhibiting multiactions (actions as sets of elementary actions) in which we can translate, in some precise sense, Milner's π-calculus. Both results will be presented in a forthcoming paper. Their implications are clear : if we can translate between OL1, OL2 and the π-calculus, we have an indication that the essential characteristics of our languages are quite close to that of the π-calculus. And this would be an indication that the correct underlying theory for object-based concurrency, with the different features we have presented in this paper, is something closer to the π-calculus than to CCS. Obviously a lot more work is required on the subject.

The main thrust of our work was motivated in part by the desire to be able to capture and characterize precisely the semantics of a language which would allow the specification of highly dynamic process structures while preserving the good specification structuring properties of a language such as CCS or CSP. With OL1, we are able to specify very dynamic structures, as was shown in our examples. However, OL1 offers very poor structuring facilities : all process structures in OL1 look essentially the same ; they are just sets of concurrent objects ; you need to look at the detail of object acquaintances (i.e. object identifiers which are known by objects) to know what interaction patterns between objects are present at a given time. Providing good structuring facilities à la CCS while preserving the dynamic properties of OL1 is something which is definitely worth investigating.

REFERENCES

[America 87] P. America : "POOL-T - A parallel object-oriented language" in: [Yonezawa 87]

[America 88] P. America, J.W. de Bakker : "Designing equivalent semantic models for process creation" - Theoretical Computer Science 60, 1988.

[ANSA 89] ESPRIT Project n°2267 (Integrated Systems Architecture) - Advanced Network Systems Architecture Reference Manual - Architecture Project Management Cambridge, UK- March 1989.

[Bal 89] H.E. Bal, J.G. Steiner, A.S. Tanenbaum : "Programming Languages for Distributed Computing Systems" - ACM Computing Surveys, Vol.21, N°3, September 1989.

[Black 87] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter : "Distribution and abstract types in Emerald" - IEEE Transactions on Software Engineering, SE 13 (1), January 1987.

[Boudol 85] G. Boudol : "Notes on Algebraic Calculi of Processes" - Advanced NATO School Series on Logics and Models for Verification and Specification of Concurrent Systems, Springer-Verlag 1985.

[Cusack 89] E. Cusack, S. Rudkin, C. Smith : "An Object-Oriented Interpretation of LOTOS" - in Proceedings FORTE 1989 - Vancouver, December 1989.

[Cusack 90] E. Cusack, M. Lai :"Object-oriented Specification in LOTOS and Z or, My Cat Really Is Object-Oriented !" - Proceedings Workshop on the Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands - 1990.

[Dasgupta 88] P. Dasgupta, R. Leblanc, W. Appelbe : "The Clouds distributed operating systems : functional description, implementation details and related work" - 8th International Conference on Distributed Computer Systems, San Jose, CA, USA. June 1988.

[Duke 90] D. Duke, R. Duke : "Towards a Semantics for Object-Z" - Proceedings VDM '90 "VDM and Z" - Lecture Notes in Computer Science, Springer-Verlag 1990.

[Ehrig 85] H. Ehrig, B. Mahr :"Fundamental of Algebraic Specification 1" - EATCS Monographs on Theoretical Computer Science - Spinger-Verlag 1985.

[Engberg 86] U. Engberg, M. Nielsen : "A Calculus of Communicating Systems with Label Passing" - Report DAIMI PB 208 - Aarhus Denmark - May 1986.

[Fiadeiro 90] J. Fiadeiro, T. Maibaum : "Describing, Structuring and Implementing Objects" - Proceedings Workshop on the Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands - May 1990.

[Goguen 90] J.A. Goguen : "Sheaf Semantics for Concurrent Interacting Objects" - Proceedings Workshop on the Foundations of Object-Oriented Languages, Noordwijkerhout, The Netherlands - May 1990.

[Hoare 1985] A. Hoare : "Communicating Sequential Processes" - Prentice Hall 1985.

[ISO 88] International Standard 8807 - "LOTOS : A Formal Description Technique Based on the Temporal Ordering of Observational Behavior" - 1988

[Liskov 88] B. Liskov, R. Scheifler : "Guardians and actions : linguistic support for robust distributed programs" - ACM Transactions on Programming Languages and Systems. Vol 5 n°3, July 1988.

[Mayr 88] T. Mayr : "Specifications of object-oriented systems in LOTOS" - In proceedings FORTE 1988.

[Meyer 88] B. Meyer : "Object-oriented Software Construction" - Prentice-Hall 1988.

[Milner 89] R. Milner : "Communication and Concurrency" - Prentice-Hall 1989.

[Milner 89a] R. Milner, J. Parrow, D. Walker : "A Calculus of Mobile Processes - Part I" - LFCS Report 89-85. University of Edinburgh June 1989.

[Milner 89b] R. Milner, J. Parrow, D. Walker : "A Calculus of Mobile Processes - Part II" - LFCS Report 89-86. University of Edinburgh June 1989.

[Milner 90] R. Milner : "Functions as Processes" - INRIA Research Report n° 1124, February 1990 - INRIA, Rocquencourt, France.

[Raj 88] R.K. Raj, E. Tempero, H.M. Levy, N.C. Hutchinson, P. Black : "The Emerald Approach to Programming" - Technical report 88-11-01 University of Washington, WA, USA - November 1988.

[Rutten 88] J.J.M.M. Rutten : "Semantic Correctness for a Parallel Object-oriented Language" - Report CS-R8843. Centrum voor Wiskunde en Informatica, Amsterdam. October 1988.

[Sernadas 89] A. Sernadas, J. Fiadeiro, C. Sernadas, H.D. Ehrig : "Abstract Object Types : A Temporal Perspective" - in Nabieqbal, Baringer and Pnueli (eds) Temporal Logic in Specification - LNCS 398 - Springer Verlag 1989.

[Wegner 87] P. Wegner : "Dimensions of Object-Based Language Design" in Proceedings OOPSLA 1987.

[Yonezawa 87] A. Yonezawa, M. Tokoro, eds : "Object-Oriented Concurrent Systems" - MIT Press 1987.