Operational Semantics and Type System for GNOME on Typed Calculus of Objects *

António Ravara
Departamento de Matemática, Instituto Superior Técnico.
Universidade Técnica de Lisboa, Portugal

Vasco Vasconcelos
Departamento de Informática,
Faculdade de Ciências da Universidade de Lisboa, Portugal
e-mail: amar@math.ist.utl.pt, vv@di.fc.ul.pt

June 96

Abstract

We present an operational semantics and a typing assignment system for the concurrent object-oriented specification language GNOME, based on a name-passing asynchronous calculus of concurrent objects, TyCO. The operational semantics is given by a map encoding GNOME classes in TyCO agents, and thus, objects (instances of classes) in processes (instances of agents). We propose a general approach to implement synchronous communication in an asynchronous process calculus, using a commit-abort protocol, achieving by this means the synchronization of all objects involved on a transaction. We treat objects with internal animation, their dynamic creation and deletion. Furthermore, the method generates a (decidable) typing assignment system that ensures lack of runtime errors for typable programs. The type of an agent certificates the communicating protocol of the corresponding class.

1 Introduction

We aim at an operational semantics and a safe type system for a very high level concurrent object-oriented specification language, called GNOME [SR94, RS95], and use a calculus of objects called TyCO (Typed Concurrent Objects) [Vas94d, Vas94c] to achieve the objective.

Process calculi has been accepted as a natural vehicle for research on the foundations of concurrent object-oriented languages [PT95]. Following [KY94], the approach is "concurrent object-oriented programming = concurrent calculus + records", a view analogous to that of research on the foundations of type systems for sequential object-oriented programming (λ-calculus + records). To give a solid operational semantics for GNOME, we choose TyCO because it is a versatile and expressive object-oriented concurrent calculus.

*Presented at the ECOOP'96 workshop on Proof-Theory of Concurrent Object-Oriented Programming

The language GNOME is an object-oriented formal specification language of concurrent, interactive, and reactive systems. It is a class-based and object-based language [Weg90], strongly typed, and with a fixed set of basic data types.

The language supports concurrency intra and inter-objects, with synchronous or asynchronous communication, dynamic creation (and deletion) of objects, inheritance with overriding, and state-dependent method invocation.

TyCO is a name-passing calculus of objects, with asynchronous communication between concurrent objects via messages carrying names. The calculus is developed along the trends of well-known models of concurrency, as the π-calculus [Mil91, MPW92], the ν-calculus [HT91], and the actor model of computation [Agh86, HBS73].

Processes are implicitly typed: no type information is present, but it can be inferred by a type system, similar to those of the λ-calculus [Bar92, HS86], those of sequential object-oriented programming (as, for example, in [Pie93, Bru95]), or of the π-calculus [Mil91]. The system assigns types to names and sets of name-type pairs (called typings) to processes, ensuring a type discipline on the use of channels, unlike the type systems that assign types to terms. Programs (processes) are statically typed.

This type system [Vas94b, Vas94d] is inductively defined by a few typing rules, one for each constructor of the calculus, with a weak rule and a subtyping rule. It is a formal one (in the sense of [HS86]) verifying properties such as subject-reduction (having as corollary that well-typed processes do not run into errors at runtime), the decidability of the typing assignment, and the computability of the typing inference. The typing of a process constitutes a communicating protocol for that process. In conclusion, it is a typing assignment system, that gives partial correctness and partial specification.

A specification in GNOME declares a set of classes, defining an object community. The agents resulting from the encoding of the classes, besides giving the operational semantics of the specification, also allow its automatic type-checking. A specification is type error free if and only if the translated agents are type error free. But for type-checking the specification directly it seems interesting to built a typing assignment system for GNOME specifications, similarly to [Vas94a].

The paper has two main parts: the presentation of the translation of GNOME in TyCO in section 2, starting with a brief description of the GNOME syntax and informal semantics, defining afterwards the syntax and the operational semantics of TyCO, and finally doing the encoding; and the construction of type system for GNOME, based on the TyCO type system in section 3. The conclusions and comparisons to related work are in section 4.

2 The translation

The operational semantics is obtained defining the semantic map of the language GNOME in TyCO (as developed in [Rav96]). We encode GNOME classes in TyCO agents, and thus, objects (instances of classes) in processes (instances of agents). The resulting agents for given GNOME classes are very elaborated and complex, but nevertheless they confirm the adequacy of the calculus of objects for expressing the operational semantics of (very high level) concurrent object-oriented languages.

2.1 The GNOME specification language

The object-oriented GNOME language is adequate for the specification of concurrent, reactive, and interactive systems. Using the terminology of Wegner [Weg90], it is a class-based and object-based language, where classes are modules for the creation of similar objects, and objects are encapsulation units. Objects have internal animation, and provide services to other objects, with synchronous or asynchronous communication. Each object is characterized by an internal state (a set of attributes) that can change during its life-cycle, either by self initiative, or by interaction with other objects. The behavior of an object is determined by its internal state (state-dependent method invocation) [SCS92]. There is concurrency intra and inter-objects, and dynamic creation (and deletion) of objects. A fragment of the language is presented by its grammar in Figure 1.

A specification is a collection of classes, each one with a public interface and a capsulated body. The interface declares the external actions (called services), with their parameters and results. In the body there are the attributes (called slots) declarations, and the methods of the internal and external actions. The methods can have four types of clauses, in the following order: its enabling condition (it can be the conjunction of several), the calling of services of other objects, the valuation of slots, and the response to the invoker of the service (only available for external actions). The semantics of a method execution will be explained later.

2.2 The Typed Calculus of Objects

The operational semantics is based on a name-passing calculus of objects, with asynchronous communication between concurrent objects via messages carrying names.

The terms of the calculus are called processes. The basic ones are objects (or servers) composed of labeled methods, and asynchronous labeled messages, targeting some object and selecting one of its labeled methods. The other constructors of the calculus are the concurrent composition of processes, and the restriction of the scope of a name to a process (dynamic creation of names). There is also a (quite convenient, but not indispensable) second syntactic category, called *agents*, denoting families of processes parameterized by sequences of names, allowing to declare an agent once and to use it several times in a given process, through a let constructor. The syntax of the calculus is presented in Figure 2.

The computational mechanism of the calculus is the interaction between an object and a message, called *message application*, that constitutes a reduction relation. The operational semantics is based on this reduction relation and on a congruence relation over processes. Both relations are defined in Figure 3.

2.3 Encoding

As the encoding is a complex task, we do it by stages: we first encode classes of passive and persistent objects, then we allow the dynamic creation and deletion of objects, and finally we add internal animation to the objects. By this methodology, one can observe the increasing complexity of the agent, expressing the semantic richness of a GNOME specification of an object community.

unit ::= `spec` *spec_id*
  type_dec*
  class*
`end spec`

type_dec ::= `type` *type_dec_id* = prim_type | *type_dec_id* | *class_id*

prim_type ::= `bool` | `nat`

class ::= `class` *class_id*
  `interface` **service***
  `body` **body**
`end` *class_id*

service ::= [`birth` | `death`] `serv` *serv_id*
  (`par` *par_id*: **type**)*
  (`res` *res_id*: **type**)*

body ::= (`slot` *slot_id*: **type**)*
  (`serv` *serv_id* **method**)*
  ([`death`] `act` *action_id* **method**)*

method ::= **enabling***
  calling*
  valuation*
  responce*

enabling ::= `enb` *condition*

calling ::= `call` *serv_id* [`as` *serv_id*] `of` *expression*:*class_id*
  (`arg` *serv_id.par_id* = *expression*)*

valuation ::= `val` *slot_id* $\ll$ *expression*

responce ::= `ret` *serv_id.res_id* = *expression*

Figure 1: The GNOME grammar

One should be aware that it is not possible to define the encoding by translating the GNOME language clause by clause. This is so because the communication between objects in GNOME is synchronous and TyCO only has asynchronous communication. Thus, we need a way to implement the atomic execution of a service, emulating synchronous communication from asynchronous. Let us first describe the semantics of a service execution: a service can only be executed if its enabling condition is true, and all the services requested by this service are enabled. As the execution of a service is atomic, all clauses should be executed "simultaneously" (only one state transition occurs). So, for the encoding we need an operator to synchronize two processes, so that the second only starts when the first gives it an explicit order (sequential composition, denoted by ;). We have:

$$P; Q \stackrel{\text{def}}{=} \nu c\ P, c \triangleright [\ enb : (\tilde{v})\ Q] \qquad (c \text{ fresh})$$

The TyCO code that implements the method of a particular service should first evaluate its enabling condition. If it is false the service is disabled, and a message is sent to communicate this to the service invoker. Otherwise, the calls to other services are tried sequentially. If any of the called services is disabled the service in question is also disabled, and again a message is sent to communicate this to the service invoker. But if all are enabled, the service is then enabled and the message returned informs the invoker of this fact. Furthermore, all objects involved on the transaction corresponding to the

Consider a contable set of names, denoted by a,b,..., and also u,v,x,y,..., a contable set of variables over agents, called agent-names, X, Y,..., and a set of labels, l,l_1,l_2, etc. Let P,Q,..., denote processes, and $\tilde{x}$ stand for a sequence of pairwise distinct names. The set of processes is given by the following grammar:

$$P \quad ::= \quad a \triangleright [\, l_1 : A_1 \,\&\, \cdots \,\&\, l_n : A_n \,] \quad | \quad a \triangleleft l : \tilde{v} \quad | \quad P, Q \quad | \quad \nu x P \quad |$$

$$X(\tilde{v}) \quad | \quad A(\tilde{v}) \quad | \quad \mathtt{let}\, X = A \,\mathtt{in}\, P$$

where A,A_1,..., denote agents, whose set is defined by the grammar:

$$A \quad ::= \quad (\tilde{x})P \quad | \quad \mathtt{rec}\, X = A$$

Figure 2: Syntax of processes

Structural congruence is the smallest congruence relation over processes generated by the following rules:

1. $P \equiv Q$, if $P \equiv_\alpha Q$;
2. $P, Q \equiv Q, P$ and $(P, Q), R \equiv P, (Q, R)$;
3. $M \equiv N$ if the method's record M is a permutation of N;
4. $\nu x P, Q \equiv \nu x (P, Q)$ if $x \notin \mathrm{fn}(Q)$;
5. $((\tilde{x})P)(\tilde{v}) \equiv P\{\tilde{v}/\tilde{x}\}$ if the lengths of $\tilde{x}$ and $\tilde{v}$ match;
6. $\mathtt{rec}\, X = A \equiv A[X := \mathtt{rec}\, X = A]$;
7. $\mathtt{let}\, X = A \,\mathtt{in}\, P \equiv P[X := A]$;

Message application represents the reception of a message by an object, with a method being selected. Let C be the communication $l_i : \tilde{v}$ of some message, and let M be the method's record $[\, l_1 : A_1 \,\&\, \cdots \,\&\, l_n : A_n \,]$ of some object; if the target of the message and the location of the object coincide then the result of the interaction between them (denoted by $M \bullet C$) is the process $P_i\{\tilde{v}/\tilde{x}_i\}$, where $A_i = (\tilde{x}_i)P_i$.

Reduction is the smallest relation over processes, generated by the following rules:

$$P' \rightarrow Q' \text{ since } P' \equiv P, P \rightarrow Q, Q \equiv Q'$$

$$\nu\tilde{x}(a \triangleright M, a \triangleleft C, \tilde{P}) \rightarrow \nu\tilde{x}(M \bullet C, \tilde{P})$$

relying that every process is structural congruent to a process of the form $\nu\tilde{x}\tilde{P}$, where $\tilde{P}$ is the concurrent composition of messages, objects and applied agent-names.

Figure 3: One-step Reduction

service execution (because they have services called) should be synchronized and locked. The object whose service was initially invoked is now waiting for the commitment of the execution (or the abortion).

Therefore, an object has two alternating states: *free*, when it is available to receive a request of one of its services, and *locked*, when it has received a service request (and answered that it was enabled, after having locked all other objects it calls), and is waiting for the commitment of the execution or for an order to abort it. This behavior corresponds to the implementation of a commit-abort protocol (see [Tel94]).

In conclusion, the agent representing a typical class of persistent and passive objects has two parameters: one for naming its instances, and other to store the value of its attribute (we consider only one attribute per class). Instances of the class represent GNOME objects in the free state: TyCO objects whose methods correspond to the services of the class, each method having two arguments: the parameter and the result of the corresponding service. When an enabled service is requested, the instance in the free state is consumed and another object representing a locked state is created, having now two methods: commit and abort. After the execution of one of them the object representing the class instance in the free state is re-created.

We present now the typical structure of this kind of classes, and the corresponding encoding we have just described:

$$
\begin{array}{l}
\llbracket \texttt{class}\ \mathrm{Class} \\
\quad \texttt{interface} \\
\qquad \texttt{serv}\ v \\
\qquad\quad \texttt{par}\ p\text{: } \mathit{type} \\
\qquad\quad \texttt{res}\ r\text{: } \mathit{type} \\
\quad \texttt{body} \\
\qquad \texttt{slot}\ s\text{: } \mathit{type} \\
\qquad \texttt{serv}\ v \\
\qquad\quad \texttt{enb}\ \mathbf{condition} \\
\qquad\quad \texttt{call}\ v_1\ \texttt{of}\ \mathbf{targ}_1\text{: } \mathrm{C}_1 \\
\qquad\qquad \texttt{arg}\ v_1.p = \mathbf{arg}_1 \\
\qquad\quad \texttt{call}\ v_2\ \texttt{of}\ \mathbf{targ}_2\text{: } \mathrm{C}_2 \\
\qquad\qquad \texttt{arg}\ v_2.p = \mathbf{arg}_2 \\
\qquad\quad \texttt{val}\ s \ll \mathbf{val\_exp} \\
\qquad\quad \texttt{ret}\ r = \mathbf{ret\_exp} \\
\texttt{end}\ \mathrm{Class}\rrbracket =
\end{array}
$$

$$
\begin{array}{l}
\texttt{rec}\ \mathrm{Class} = (\mathit{self}\ s) \\
\mathit{self} \triangleright [\ v : (p\ r) \\
\qquad \texttt{if}\ \llbracket \mathbf{condition} \rrbracket\ \texttt{then} \\
\qquad\quad \nu x_1\ \llbracket \mathbf{targ}_1 \rrbracket \triangleleft v_1 : \llbracket \mathbf{arg}_1 \rrbracket\ x_1, \\
\qquad\qquad x_1 \triangleright [\mathit{enb} : (v_1)\ \nu x_2\ \llbracket \mathbf{targ}_2 \rrbracket \triangleleft v_2 : \llbracket \mathbf{arg}_2 \rrbracket\ x_2, \\
\qquad\qquad\quad x_2 \triangleright [\ \mathit{enb} : (v_2)\ \nu t\ r \triangleleft \mathit{enb} : t, \\
\qquad\qquad\qquad \mathrm{CommitAbort}(t\ \mathit{self}\ v_1\ v_2\ s\ p) \\
\qquad\qquad\quad\ \ \&\ \mathit{dis} : v_1 \triangleleft \mathit{abort}\ ; \\
\qquad\qquad\qquad (r \triangleleft \mathit{dis}, \mathrm{Class}(\mathit{self}\ s))\ ] \\
\qquad\qquad \&\ \mathit{dis} : r \triangleleft \mathit{dis}, \mathrm{Class}(\mathit{self}\ s)\ ] \\
\qquad \texttt{else}\ r \triangleleft \mathit{dis}, \mathrm{Class}(\mathit{self}\ s)\ \texttt{fi}]
\end{array}
$$

where CommitAbort is:

$$
\begin{array}{l}
\mathrm{CommitAbort} = (\mathit{self}\ c\ v_1\ v_2\ s\ p) \\
\mathit{self} \triangleright [\mathit{commit} : (r)\ v_1 \triangleleft \mathit{commit} : \llbracket \mathbf{arg}_1 \rrbracket\ ; v_2 \triangleleft \mathit{commit} : \llbracket \mathbf{arg}_2 \rrbracket\ ; \\
\qquad\qquad (r \triangleleft \mathit{enb} : \llbracket \mathbf{ret\_exp} \rrbracket, \mathrm{Class}(c\ \llbracket \mathbf{val\_exp} \rrbracket)) \\
\quad \&\ \mathit{abort} : (r)\ v_1 \triangleleft \mathit{abort}\ ; v_2 \triangleleft \mathit{abort}\ ; (r \triangleleft \mathit{enb}, \mathrm{Class}(c\ s))]
\end{array}
$$

Moving forward, we treat the dynamic creation and deletion of objects; it is necessary to know the identities of the existing instances of each class. Thus, with a class we associate a manager of its instances, with a list of the identifiers of those instances (a data structure other than a list could be used). So, it is possible to create and delete objects of a given class, and to ask about the existence of a particular object, by putting (new) names in the list, removing names from the list, and looking for names in the list. Each class has an associated manager (a particular object) with three methods: *birth*, to create instances, *death*, to delete them, and *exists*, to check whether a given instance is alive.

The instances of a class have the structure and behavior described in the previous paragraphs.

In order to have classes of active objects, we finally encode internal animation (private actions, executed nondeterministically). The process representing a GNOME object is the parallel composition of two "engines" (objects): one receiving and queuing demands for services (one method for each service), and another choosing randomly (with a uniform distribution) the next activated internal actions or invoked services should be executed (only one service, called internally). The first engine is accessible from the "environment", constituting the interface of the object, whereas the second is the manager of the object animation, responsible for changing its internal state, and it is obviously capsulated.

To illustrate this architecture, we encode now an iterative version of the factorial:

$[\![$`class` Fact
  `interface`
    `birth` `serv` req
      `par` n: `nat`
    `death` `serv` ans
      `res` $n!$: `nat`
  `body`
    `slot` n: `nat`
    `slot` $n!$: `Fact`
    `serv` req
      `val` $n \ll$ req.n
      `val` $n! \ll 1$
    `serv` ans
      `enb` $n = 1$
      `ret` ans.$n! = n!$
    `act` step
      `enb` $n > 1$
      `val` $n! \ll n{*}n!$
      `val` $n \ll n$-1
`end` Fact$]\!] =$
$(g\ c)\ \nu l\ \mathrm{Fact}(g\ c\ l), \mathrm{NewSet}(l)$

This is the agent representing the class: NewSet is an agent representing an empty set, with services to put, take and verify the existence of an element in the set, and the class manager is an instance of the agent Fact; an instance of the class is created as an instance of the agent NewF.

The TyCO code of the agents is as follows:

$$
\begin{aligned}
&\mathrm{Fact} = (\mathit{self}\ c\ l)\ \mathtt{rec}\ \mathrm{Fact}\\
&\mathit{self} \rhd [\ \mathit{birth} : (n\ r)\quad \nu i\ r \lhd \mathit{enb} : i, \mathrm{NewF}(i\ \mathit{self}\ g\ n\ l), \mathrm{Fact}\\
&\qquad \&\ \mathit{death} : (i\ r)\, l \lhd \mathit{pop} : i\,; (r \lhd \mathit{enb}, \mathrm{Fact})\\
&\qquad \&\ \mathit{exists} : (i\ r)\, \mathtt{if}\ l \lhd \mathit{exists} : i\ \mathtt{then}\ r \lhd \mathit{enb}, \mathrm{Fact}\\
&\qquad\qquad\qquad \mathtt{else}\ r \lhd \mathit{dis}, \mathrm{Fact}\ \mathtt{fi}\ ],\\
&c \rhd [\ \mathit{req} : (n\ r) \mathit{self} \lhd \mathit{birth} : n\ r, \mathrm{Fact}\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{NewF} = (\mathit{self}\ c\ g\ p\ l)\\
&\mathit{self} \rhd [\ \mathit{commit} : (r)\ \ \nu i\ l \lhd \mathit{push} : i\,; (r \lhd \mathit{enb}, \mathrm{InstF}(i\ g\ p\ [\![1]\!]))\\
&\qquad \&\ \mathit{abort} : (r)\ r \lhd \mathit{enb}\ ]
\end{aligned}
$$

An instance of InstF is an object of the class Fact, where:

$$
\begin{aligned}
\mathrm{InstF} = (\mathit{self}\ g\ n\ n!)\ \nu\ \mathit{ans}\ a\ \mathit{ae}\ \mathit{ai}\ f\ o\ &\mathrm{Interface}(\mathit{self}\ \mathit{ae}\ \mathit{ans}\ f),\\
&\mathrm{Anim}(a\ \mathit{ae}\ \mathit{ai}\ f\ o), \mathrm{Ae}(\mathit{ae}\ a\ \mathit{ai}\ g\ \mathit{self}), \mathrm{Ai}(\mathit{ai}\ a),\\
&a \lhd \mathit{go} : n\ n!, \mathrm{Random}(o), \mathrm{List}(f)
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Interface} = (\mathit{self}\ \mathit{ae}\ v\ f)\ \mathtt{rec}\ \mathrm{Int}\\
&\mathit{self} \rhd [\ \mathrm{ans} : (r)\ f \lhd \mathit{enqueue} : v\ r\,; \mathrm{Int}\\
&\qquad \&\ \mathit{kill} : (o)\ \mathtt{if}\ o = \mathit{ae}\ \mathtt{then}\ f \lhd \mathit{clean}\ \mathtt{else}\ \mathrm{Int}\ \mathtt{fi}\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Anim} = (\mathit{self}\ \mathit{ae}\ \mathit{ai}\ f\ o)\ \mathtt{rec}\ \mathrm{Anim}\\
&\mathit{self} \rhd [\ \mathit{go} : (s_1\ s_2)\\
&\qquad\qquad \mathtt{case}\ o \lhd \mathit{val} : [\![2]\!]\ \mathtt{of}\\
&\qquad\qquad\quad 1 :\ \mathtt{case}\ f \lhd \mathit{dequeue}\ \mathtt{of}\\
&\qquad\qquad\qquad\quad \mathit{dequeued} : (l\ r)\ \mathit{ae} \lhd l : s_1\ s_2\ r\\
&\qquad\qquad\qquad\quad \mathit{empty} : \mathit{self} \lhd \mathit{go} : s_1\ s_2\\
&\qquad\qquad\qquad \mathtt{esac}\\
&\qquad\qquad\quad 2 :\ \mathit{ai} \lhd \mathit{step} : s_1\ s_2\\
&\qquad\qquad \mathtt{esac}, \mathrm{Anim}\\
&\qquad \&\ \mathit{kill} :\ \mathbf{0}\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Ai} = (\mathit{self}\ a)\ \mathtt{rec}\ \mathrm{Ai}\\
&\mathit{self} \rhd [\ \mathit{step} : (n\ n!)\\
&\qquad\qquad \mathtt{if}\ [\![n > 1]\!]\ \mathtt{then}\ a \lhd \mathit{go} : [\![n * n!]\!]\ [\![n-1]\!]\\
&\qquad\qquad \mathtt{else}\ a \lhd \mathit{go} : n\ n!\ \mathtt{fi}, \mathrm{Ai}\\
&\qquad \&\ \mathit{kill} :\ \mathbf{0}\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{Ae} = (\mathit{self}\ a\ \mathit{ai}\ g\ i)\ \mathtt{rec}\ \mathrm{Ae}\\
&\mathit{self} \rhd [\ \mathit{ans} : (n\ n!\ r)\\
&\qquad\qquad \mathtt{if}\ [\![n = 1]\!]\ \mathtt{then}\ \nu o\ r \lhd \mathit{enb} : o, \mathrm{Ans}\\
&\qquad\qquad \mathtt{else}\ r \lhd \mathit{dis}, \mathrm{Ae}\ \mathtt{fi}\ ]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Ans} = \mathit{self} \rhd [\ &\mathit{commit} : (r)\ g \lhd \mathit{death}; (r \lhd \mathit{enb} : n!, a \lhd \mathit{kill}, \mathit{ai} \lhd \mathit{kill}, i \lhd \mathit{kill})\\
&\&\ \mathit{abort} : (r)\ r \lhd \mathit{enb}, a \lhd \mathit{go} : n\ n!, \mathrm{Ae}\ ]
\end{aligned}
$$

The verification of the correctness of the encoding is very simple: one can prove by induction that both programs calculate the same values, with the same transitions on n and $n!$.

This encoding can be a general approach to define operational semantics based on concurrent calculus of objects for concurrent object-oriented languages. The commit-abort protocol implements synchronous communication from asynchronous messages and an operator of sequential composition. The class manager is adequate for class-based and object-based object-oriented languages with dynamic creation (and deletion) of objects, and the architecture of an active object can basically be the parallel composition of the two "engines" (internal and external animation). Locally to each object we have treated *fairness*: no asked service will be postponed forever and every internal action will eventually be executed, as one can easely confirm observing the behavior of the agent Anim: it is

The typing assignment system ($\mathbf{TS}_{\mu\forall}$) is inductively defined by the following typing rules:

VAR $B \cdot X : \tilde{\alpha} \vdash X \blacktriangleright \tilde{\alpha}, \emptyset \in \mathbf{TS}_{\mu\forall}$

$\approx$ $B \vdash P \blacktriangleright \Gamma \cdot x : \beta \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{TS}_{\mu\forall}$ and $\alpha \approx \beta$

MSG$_{\mu\forall}$ $B \vdash a \lhd l_1 : \tilde{v} \blacktriangleright \{a : \beta\} \cup \tilde{v} : \tilde{\alpha}_1 \in \mathbf{TS}_{\mu\forall}$ provided that $\beta \approx [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]$

OBJ $B \vdash a \rhd [l_1 : A_1 \ \& \ \cdots \ \& \ l_n : A_n] \blacktriangleright \{a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n]\} \cup \Gamma_1 \cup \cdots \cup \Gamma_n \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash A_1 \blacktriangleright \tilde{\alpha}_1, \Gamma_1, \ldots, B \vdash A_n \blacktriangleright \tilde{\alpha}_n, \Gamma_n \in \mathbf{TS}_{\mu\forall}$ and $a : [l_1 : \tilde{\alpha}_1, \ldots, l_n : \tilde{\alpha}_n] \asymp \Gamma_1 \asymp \cdots \asymp \Gamma_n, n \geq 0$

SCOP $B \vdash \nu x \ P \blacktriangleright \Gamma \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{TS}_{\mu\forall}$

COMP $B \vdash P, Q \blacktriangleright \Gamma \cup \Delta \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash P \blacktriangleright \Gamma, B \vdash Q \blacktriangleright \Delta \in \mathbf{TS}_{\mu\forall}$ and $\Gamma \asymp \Delta$

ABS $B \vdash (\tilde{x})P \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash P \blacktriangleright \Gamma \cdot \tilde{x} : \tilde{\alpha} \in \mathbf{TS}_{\mu\forall}$

WEAK $B \vdash P \blacktriangleright \Gamma \cdot x : \alpha \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash P \blacktriangleright \Gamma \in \mathbf{TS}_{\mu\forall}$ and $x \notin \Gamma$

APP $B \vdash U(\tilde{v}) \blacktriangleright \Gamma \cup \{\tilde{v} : \tilde{\alpha}\} \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash U \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{TS}_{\mu\forall}$, where U is an agent or an agent-name, and $\Gamma \asymp \{\tilde{v} : \tilde{\alpha}\}$

LET $B \vdash \texttt{let}\ X = A\ \texttt{in}\ P \blacktriangleright \Gamma \cup \Delta \in \mathbf{TS}_{\mu\forall}$ provided that $B \vdash A \blacktriangleright \tilde{\alpha}, \Gamma, B \cdot X : \tilde{\alpha} \vdash P \blacktriangleright \Delta \in \mathbf{TS}_{\mu\forall}$ and $\Gamma \asymp \Delta$

REC $B \vdash \texttt{rec}\ X = A \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{TS}_{\mu\forall}$ provided that $B \cdot X : \tilde{\alpha} \vdash A \blacktriangleright \tilde{\alpha}, \Gamma \in \mathbf{TS}_{\mu\forall}$

$\forall$-INTRO $B \vdash A \blacktriangleright \forall t. \sigma, \Gamma \in \mathbf{ST}_{\mu\forall}$ provided that $B \vdash A \blacktriangleright \sigma, \Gamma \in \mathbf{TS}_{\mu\forall}$ and t is bound in Γ and/or in B

$\forall$-ELIM $B \vdash A \blacktriangleright \sigma[\alpha/t], \Gamma \in \mathbf{ST}_{\mu\forall}$ provided that $B \vdash X \blacktriangleright \forall t. \sigma, \Gamma \in \mathbf{TS}_{\mu\forall}$

In the last two rules σ stands for a polymorphic type. It can replace the monomorphic type $\tilde{\alpha}$ in the other rules, and in bases (but not in typings).

Figure 4: TyCO typing assignment system

randomly decided (with uniform distribution) if it should be executed an internal action or an asked service.

3 The typing assignment system for GNOME

As we said before the GNOME language is strongly typed, but we aim a type system with the set of properties of the TyCO typing assignment system. The objective is to guarantee the absence of "message not understood" errors: no object will ever receive a message not suitable for it, either because the invoked service doesn't exist or because the arguments don't match.

There are two approaches for type-checking a GNOME specification: type-checking the corresponding TyCO code, or directly type-checking the specification by building to GNOME a typing assignment system from the encoding and the TyCO type system (Figure 4). We present this second alternative.

The procedure is as follows: starting from the type information present in the specification, we set as axioms the assignments of types to the attributes, and to the parameters and results of the services. These axioms are added to the TyCO typing assignment system: we use them to try to infer a type for all the expressions present in the specification assuming there is an encoding in TyCO of the GNOME types. These expressions can be logical, arithmetic, or attribute names. If all the expressions are typable (accordingly to the axioms and the TyCO encoding of the GNOME types), and if it is also typable the assignments of values, then the classes are well-typed. In this case it is possible to directly infer their types: they will be records whose fields have the service names, and for each field there is a type expressing the functional behavior of that service (arguments type and result type).

To present the typings of the three encodings developed in the previous section let us consider that the result of a service, once it is carried in a message, has a type:

$$\tau = [dis,\, enb : [abort : [enb], commit : [enb : \texttt{res\_type}]]]$$

This type clarifies that the caller of the service must wait for information about its enabledness (a *disabled* or *enabled* message). In this last case, the instance to which the service was called is locked, and must wait for an order to abort or to commit the execution. In both cases it will send a message *enb* to release the caller. If the service was executed, this message carries the result of the execution.

We can now present the final types: For the first encoding, the type of the instances of a class with only one service v is:

$$\iota = \ [v : \texttt{par\_type} \rightarrow \tau]$$

The class has two parameters, the first for naming its instances, and the second representing the attribute, and so has type: ι `slot_type`.

For the second and the third encondings, the types are the same because active and passive objects have the same interface. The types of the main entities are:

set of instances type: [*exists*: $\iota \rightarrow$[*false*,*true*], *add*: $\iota \rightarrow$[*enb*], *rem*: $\iota \rightarrow$[*enb*]]

The first method returns a boolean; the other two return a synchronization message after modifying the set.

class manager type: [*birth*: `par_type` $\rightarrow \iota$, *death*: $\iota \rightarrow$[*enb*], *exists*: $\iota \rightarrow$[*dis*,*enb*]]

The first method creates an instance of the class, the second deletes it and send a synchronization message, and the third one looks for an instance and send an enabling message.

class instance type: ι

A record whose fields are labeled with the services names, and whose arguments are the type of the parameter it accepts and the type of the message where the result is sent.

The information given by these types is richer than the one in the specification interface: the semantics of a dynamic object is clarified by its type.

The encoding of the basic types (corresponding to the simple GNOME types) treats operations as total functions, as usual, and as such, a class of runtime errors is not caught by the type system. For example, the predecessor of zero is zero. Because of this, the system can't be used to prove invariance properties about the specification, even if it seems that all it is needed is there, as shown in [Rav96]. We want to see what changes should be made to the typing assignment system so this would be possible.

4 Conclusions and related work

We developed a general method to define operational semantics based on concurrent object-oriented calculus to high level object-oriented programming and specification languages. This has been done before (for example, in [KY94, Vas94a]) and it is know accepted as an approach to study foundations of concurrent object-oriented languages [PT95]. But we do it for a much more complex language, as it is GNOME, demanding a structured development of the semantics, that can constitute a general method. The typing assignment systems is given for free guaranteeing the safety of the programs, and the types give a communicating protocol with some semantic information. The development of more flexible type systems that take some account of the structure and the behavior of the programs is an interesting problem, addressed now by different perspectives, as for example in [AGN95, Nie95, LW95, Pie95].

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, 1986.

[AGN95] S. Abramsky, S. Gay, and R. Nagarajan. Specification structures and propositions-as-types for concurrency. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure vs. Automata - Proceedings of the VIIIth Banff Higher Order Workshop.* Springer-Verlag LNCS, 1995.

[Bar92] H. Barendregt. Lambda-calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

[Bru95] K. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Research report, Williams College, U.S.A., 1995.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. A universal, modular actor formalism for artificial intelligence. In *3th International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[HS86] J. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume LNCS 512, pages 141–162. Springer-Verlag, 1991.

[KY94] N. Kobayashi and A. Yonezawa. Typed-theorectical foundations for concurrent object-oriented programming. Technical report, Department of Information Science, University of Tokyo, Japan, 1994.

[LW95] X. Liu and D. Walker. A polymorphic type system for the polyadic π-calculus. Research report, Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K., 1995.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical report ECS-LFCS 91-180, University of Edinburgh, U. K., 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i e ii. *Information and Computation*, 100:1–77, 1992. Also available as Technical Report ECS-LFCS 89-85/86, University of Edinburgh.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[Pie93] B. Pierce. Simple type-theorectic foundations for object-oriented programming. *Journal of Functional Programming*, 1993.

[Pie95] B. Pierce. Programming in the π-calculus: an experiment in concurrent language design. Tutorial notes for PICT version 3.6k, Computer Laboratory, University of Cambridge, U. K., 1995.

[PT95] B. Pierce and D. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice in Parallel Programming*, volume LNCS 907, pages 187–215. Springer-Verlag, 1995.

[Rav96] A. Ravara. An operational semantics for the language GNOME based on typed concurrent objects. Master's thesis, IST, Universidade Técnica de Lisboa, 1996. Supervised by A. Sernadas and V. Vasconcelos. In Portuguese.

[RS95] J. Ramos and A. Sernadas. A brief introduction to GNOME. Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995.

[SCS92] A. Sernadas, J. F. Costa, and C. Sernadas. Object specification with diagrams: the OBLOG approach. Research report, Section of Computer Science, Departament of Mathematics, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1992. Awared the Descartes Prize by the Data Processing Institute of the Ministry of Finances of Portugal.

[SR94] A. Sernadas and J. Ramos. The GNOME language: Syntax, semantics and calculus. Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1994. In Portuguese.

[Tel94] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[Vas94a] V. Vasconcelos. An operacional semantics and a typing system for ABCL/1 based on a calculus of objects. Technical report CS 94-001, Keio University, Japan, 1994.

[Vas94b] V. Vasconcelos. Predicative polymorphism in π-calculus. In *6th Parallel Architectures and Languages Europe*, volume LNCS 817, pages 425–437. Springer-Verlag, 1994.

[Vas94c] V. Vasconcelos. *A Process-Calculus Approach to Typed Concurrent Objects*. PhD thesis, Department of Computer Science, Keio University, Japan, 1994.

[Vas94d] V. Vasconcelos. Typed concurrent objects. In *8th European Conference on Object-Oriented Programming*, volume LNCS 821, pages 100–117. Springer-Verlag, 1994.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS MESSENGER*, 1(1):7–87, 1990.