An Algebraic Semantics for GNOME via a Translation to ÉTOILE Specifications*

Marc Aiguier†
Gilles Bernot†
Jaime Ramos‡
Amílcar Sernadas‡

August 20, 2008

Abstract

GNOME is a simplified and revised version of the object oriented specification language OBLOG. A formal semantics based on temporal logic has already been defined, and alternative semantics are also being studied. The goal of this article is to propose an algebraic semantics for GNOME, using ÉTOILE.

ÉTOILE is an algebraic theory for the specification of systems, with an object oriented specification style. It allows to specify objects with local states, systems and their invariants. There is also in ÉTOILE a logical system which is sound w.r.t. the algebraic semantics.

Given a GNOME specification, we obtain an algebraic semantics for it by translating the GNOME specification into an ÉTOILE specification. The main difficulties come from the fact that GNOME is a concrete specification language with built-in primitives whilst ÉTOILE is only a specification theory, and also from the way methods are called and executed in GNOME.

Proofs can be performed from a GNOME specification using the ÉTOILE calculus.

Keywords: Object Oriented specifications, Algebraic semantics, Specification languages, Formal proofs

1 Introduction

The work reported in this article is a proposal to give an algebraic semantics to GNOME (developed in Lisbon). For this purpose we use the new ÉTOILE algebraic formalism (developed in Evry).

GNOME [SR94] is a simplified and revised version of the object-oriented specification language OBLOG [SGS92, SGG+92, SCS92, SRGS91, ESD93, CSS89]. As OBLOG it supports classes of objects with associated methods and attributes, dynamic object creation and deletion, inheritance with overriding and some aditional features, such as state-dependent calling. TROLL [JSS91, JSHS91, Jun93, JSHS95] is another noteworthy dialect of OBLOG.

*This work was partly supported by CEC under ESPRIT-III BRA WG 6071 IS-CORE and WG6112 COMPASS.

†Laboratoire de Mathématiques et d'Informatique, Université d'Evry, Cours Monseigneur Romero, 91025 Evry cedex, France. E-mail: {aiguier,bernot}@lami.univ-evry.fr.

¶Departmento de Matemática, Instituto Superior Técnico, Av. Rovisco Pais, 1000 Lisboa, Portugal. E-mail: {jabr,acs}@math.ist.utl.pt.

A formal semantics for GNOME, based on temporal logic, has been defined in [Ram95] on an enrichment of OSL [SSC95, SS94]. This includes a calculus for reasoning about GNOME specifications, with special emphasis on safety and response properties. A compiler of a significant fragment of the language is also available [CP94].

The ÉTOILE specification formalism [Aig95a, AB95] is an algebraic theory which has been developed to keep a specification as abstract as possible. The ÉTOILE syntax has been designed in order to facilitate the description of the *behaviour* of methods, and the object states are left as much implicit as possible in the specifications [ABBI94]. The specification style is as close as possible to functional style, in the spirit of classical algebraic specifications [GTW78, EM85]. Nevertheless, since the objects have states that can evolve, "interactive" primitives have been introduced such as "**;**", to handle sequentiality, or the "**alive**" predicate to specify the existence of an object.

One of the advantages of this work has been to test (and improve) the expressive power of ÉTOILE: to be able to translate a significant part of an already recognized specification language is a good experiment. Moreover, to give several alternative semantics to GNOME is one step more in the long process of unifying the various Object Oriented approaches, and a language like GNOME can play the role of a catalyst with that respect.

The solution we follow in this article is a translation from GNOME into ÉTOILE.

The difficulties raised by our "translation approach" result from the fact that GNOME is a concrete specification *language* (with public and private methods, encapsulation, inheritance and other built-in primitives) while ÉTOILE is only a specification *theory* without such primitives. For instance, there is nothing in ÉTOILE to distinguish private operations from public ones (however induction proofs have to be performed with respect to visible methods only).

After an informal presentation of GNOME is section 2 and of ÉTOILE in section 3 we describe the translation in section 4. Section 5 shows how this translation can be used to prove some properties using the ÉTOILE calculus.

2 Informal presentation of GNOME

There are in GNOME two types of actions: external (or public) actions (also called services) and internal (or private) actions. An external action of an object can only *happen* when called by some other object. An internal action of an object, on the other hand, cannot be called by another object and it may *happen* whenever it is enabled. There is still another possible way for objects to interact in GNOME, via event broadcasting. However this will not be detailed here.

There is also in GNOME a new primitive for encapsulation: the *region*. In a region (which is defined within a class) we can specify other classes, types, events, etc. . . , but these will be *encapsulated*, i.e., will only be *available* for the instances of the class where we defined the region and for the classes defined within that region. The region is specified in a different module and its existence is indicated by the key word `external` in the body of the owner class.

As to data types, GNOME has a fixed set of basic data types (e.g. integers, booleans, lists. . .), but, in opposition to OBLOG, does not have any specific constructors for data type specification.

2.1 Gnome by example

In this section we present the syntax of GNOME through some examples and at the same time give some hints about the semantics. Two other examples, without comments, are included as appendices.

Example 2.0.1 (Producer-Consumer) *Assume we want to specify a system which manipulate a set of triplets made of a producer, a consumer and a buffer with unbounded capacity. In the next page we present a* GNOME *specification of such a system.*

A class in GNOME has two main components: the `interface` and the `body`. In the interface, we declare the public actions (`ser`vices) and their arguments There are two types of arguments: *input* arguments (also called `par`ameters) and *output* arguments (also called `res`ults). In the body we declare the attributes of the class (`slot`s), the internal `act`ions and define the methods of all actions. A method can have four types of clauses: `ena`bling conditions; `call`ing clauses; `val`uations and `ret`urn clauses (this last clause can only be used in the method of a service). The evaluation of a method in GNOME is atomic. Consider one action, for instance the action *consume* of *Consumer*. The evaluation of the corresponding method consists in evaluating the enabling conditions (in the case of *consume* there are none):

- if the conditions hold:
  - call the services according to the calling clauses (in the present example, the service *get* of the instance *buf* of class *Buffer*, which triggers the execution of the associated method). The keyword `of` in each calling clause identifies the object being called, and the keyword `arg` is used to actualize the parameters within a `call`;
  - evaluate the expressions on the valuation clauses (in the present example, the expression c);
  - assign the result of the evaluations to the corresponding slots, (via `<<`);
- otherwise the action is not enabled and cannot be taken. If one of the called services is not enabled the method of the action is not enabled and cannot be executed (in the present example, if *get* is not enabled the method of *consume* cannot be executed).

Notice that the evaluation of the valuation clauses is simultaneous thus the final result is independent of the order of these clauses. The same applies to the other clauses.

As the classes *Producer*, *Consumer* and *Buffer* are defined within the body of the class *ProdConsum* they are encapsulated. For instance, the services of an instance of class *Buffer* can only be called by the instance of *ProdConsum* where it was created or by the instance of *Producer* or *Consumer* that were created at the same time.

```
spec ProducerConsumer
...
class ProdConsum
  interface
    birth serv create
    serv produce
    serv consume
  body
    external
end ProdConsum

region of ProdConsum for ProducerConsumer
  body
    slot prod:Producer
    slot cons:Consumer
    slot buf:Buffer
    serv create
      call new of next:Buffer
      call new of next:Producer
        arg new.buf=Buffer.next
      call new of next:Consumer
        arg new.buf=Buffer.next
      val buf<<Buffer.next
      val prod<<Producer.next
      val cons<<Consumer.next
    serv produce
      call produce of prod:Producer
    serv consume
      call consume of cons:Consumer

class Buffer
  interface
    birth serv new
    serv put par item:nat
    serv get res item:nat
  body
    slot contains:list(nat)
    der slot size:nat by  legth(contains)
    serv new
      val contains<<newlist
    serv put
      val contains<<append(contains,put.item)
    serv get
      enb not size=0
      val contains<<tail(contains)
      ret get.item=head(contains)
end Buffer

Producer
  interface
    birth serv new par buf:Buffer
  body
    slot mybuf:Buffer
    slot nbprod:nat
    serv new
      val mybuf<<new.buf
      val nbprod<<0
    act produce
      call put of mybuf:Buffer
        arg put.item=0
          the message is not relevant
      val nbprod<<nbprod+1
end Producer

class Consumer
  interface
    birth serv new par buf:Buffer
  body
    slot mybuf:Buffer
    slot nbcons:nat
    serv new
      val mybuf<<new.buf
      val nbcons<<0
    act consume
      call get of mybuf:Buffer
          we do not use the return
            value of get
      val nbcons<<nbcons+1
end Consumer

end reg
end spec
```

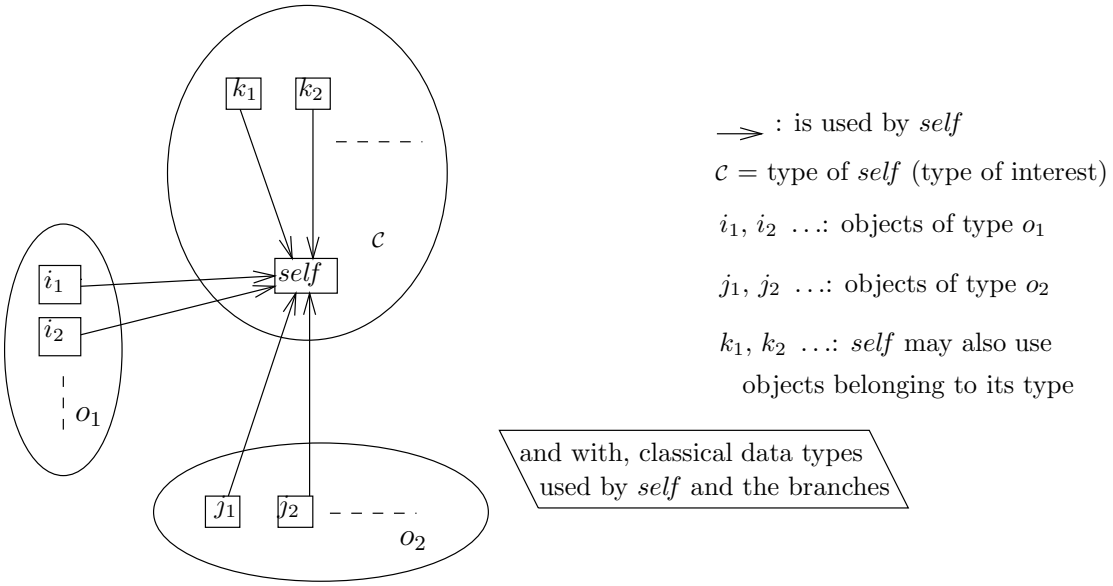
3 Informal presentation of "ÉTOILE" algebraic specifications

This section briefly outlines the formalism of ÉTOILE specifications, which is more rigorously described in [AB95]. There are two levels of specification:

- object type specifications (Section 3.1), that describe the behaviour of an object class independently of any system which might surround it,
- system specifications (Section 3.2), that contain several object type specifications and specify the behaviour of a system, using them.

3.1 Object type specifications

Roughly speaking, a class is a set of objects sharing common behaviours [EGS91]. Thus, to specify these common behaviours (i.e., an object type), it is sufficient to specify one of these objects, conventionally called *self*. Moreover an object can use services from other objects. Consequently, an object type specification describes "a view" (the one of *self*) like a star (*étoile* in French), the center of which is the object *self* and whose branches are these objects. Intuitively, *self* is fully specified while the branches only outline the functionalities used by the center, *as they are seen by self*. The branches play the role of parameters, they are not fully described, and their signatures are not exhaustive. In the classical algebraic modular approaches, such parameter parts are also often distinguished in a module [BEPP87, EGR94, NOS95].



An object type specification is defined by a signature (Section 3.1.1) and a set of formulas (Section 3.1.3). Examples can be found in Section 4 and in Appendix A.

3.1.1 Signature

Definition 3.1 *A "ÉTOILE-set" is a triplet* $\mathcal{S} = (\mathcal{C}, \mathcal{O}, \mathcal{D})$ *where* O *and* D *are sets,* $\mathcal{C} \notin D$, *and* $O \cap D = \emptyset$.

$\mathcal{S}$ is the type part of the view of *self* pictured above. O is the set of object types used by *self*. $\mathcal{C} \in O$ is allowed. Intuitively, an element i of type o, with $o \in O$, will be an object identity,

and D is the set of classical data types. To get a signature, it remains to have operations, here functions and methods.

Definition 3.2 *An* object type signature Θ *is a triplet* $\langle \mathcal{S}, \mathcal{F}, \{\mathcal{M}^\imath\}_{\imath \in \{\mathcal{C}\} \cup \mathcal{O}} \rangle$ *where:*

- $\mathcal{S}$ *is a* ÉTOILE*-set.*
- F *is a set of function names with an arity of the form* $(s_1 \cdots s_n \to s)$ *where* $s_i \in S$ *and* $s \in S$*.*
- *for each* o*,* M^o *is a set of method names with an arity of the form* $(s_1 \cdots s_n \to [s])$ *where* $s_i \in S$ *and* $s \in S$ *(the notation* $[s]$ *means that* s *is optional*[1]*).*

Roughly, "ÉTOILE-algebras" contain:

- usual data sets indexed by D
- identity sets indexed by O
- local state sets indexed by (a copy of) $\{c\} \cup O$; moreover, to introduce dynamic aspects, there is a preorder on local states which rigorously controls the side effects induced by the methods.

Intuitively, identity sets can also be seen as supplementary data sets, and states are used as "semantic modifiers" of the method semantics. Functions in F are classical operations as for usual abstract data types ([GTW78, EM85]). Their semantics will not depend on object states, nor modify them. By default, the methods in M^o can be performed by any object of type o and their semantics will depend on (and will modify) the object states. Contrarily to several approaches [EDS93, FCSM91, GD92, Dau92, DG94], we do not distinguish methods which actually modify states (e.g., actions) from those which only observe them (e.g., attribute observers) and we do not distinguish methods which create or delete objects. These properties are dynamic and will be specified by axioms.

3.1.2 Terms

Let $\Theta = \langle \mathcal{S}, \mathcal{F}, \{\mathcal{M}^\imath\}_{\imath \in \{\mathcal{C}\} \cup \mathcal{O}} \rangle$ be an object type signature and V be a S-indexed set of variables.

- Of course, every variable is a term, and we admit all the usual terms of the form $f(t_1, \cdots, t_n)$ where $f : s_1 \cdots s_n \to s$ belongs to F and t_i are terms of sort s_i.
- For a method $m : s_1 \cdots s_n \to [s]$ belonging to M^o we have to identify the object which is supposed to perform it. We write terms of the form: $(t.m(t_1, \cdots, t_n))$ where t_i are terms of sort s_i and t is a term of sort o. (The term t denotes the identity of the object supposed to perform m, possibly *self* if $o = c$, in which case we simply write $m(t_1, \cdots, t_n)$ by notation abuse.)

Roughly, the evaluation of a term in a ÉTOILE-algebra is non-deterministic [WM95]. It is based on a non-deterministic bottom-up strategy with state evolutions. Moreover, since we consider implicit states, the order in which terms are performed is significant. Thus, we

[1] A method with an arity where s is missing can be seen as a simple procedure.

also consider terms of the form: $(t_1 \,;\, \cdots \,;\, t_n)$ where the t_i are well-formed terms as above. Intuitively such a term denotes the values of its last term (i.e., t_n) after all the t_i have been performed sequentially.

Lastly, the term $(t_1 \,;\, \cdots \,;\, t_n)_{\downarrow_t}$ denotes the state of an object identified by t after the sequence $(t_1 \,;\, \cdots \,;\, t_n)$ has been performed. t can be *self*, in which case we simply write $(t_1 \,;\, \cdots \,;\, t_n)_{\downarrow}$ by notation abuse.

3.1.3 Formulas

There are two kinds of equational atoms:

- $(t_1 \,;\, \cdots \,;\, t_n) = (u_1 \,;\, \cdots \,;\, u_m)$ where t_n and u_m have the same sort $s \in S$.
  Intuitively, an instance of such an atom is satisfied if t_n and u_m can get a unique equal value after the t_i (resp. the u_j) have been performed.

- $(t_1 \,;\, \cdots \,;\, t_n)_{\downarrow_t} = (u_1 \,;\, \cdots \,;\, u_m)_{\downarrow_u}$ where t and u have the same sort $o \in O$.
  Intuitively, an instance of such an atom is satisfied if the state of the object t after the t_i have been performed is the same as the state of u after the u_i have been performed.

There are also three unary predicates to build atoms:

- **succeed**(t) means intuitively that the calculation of t always finishes (no deadlock) whatever the state evolution is.

- **wait**(t) means intuitively that the calculation of t never finishes (it always raises a deadlock) whatever the state evolution is.

- **alive**(t), where the type of t must belong to O, means that the objects identified by the values of t do exist.

Notice that **succeed**(t) implies $\neg$ **wait**(t) (or **wait**(t) implies $\neg$ **succeed**(t)) but they are *not* equivalent. There are terms t whose evaluation can finish or not depending on the underlying state evolutions, in that case neither **succeed**(t) nor **wait**(t) is satisfied.

The formulas are inductively defined from the atoms above, usual connectives belonging to $\{\neg, \wedge, \vee, \cdots\}$ and usual quantifiers belonging to $\{\forall, \exists\}$. Moreover, the notion of implicit states induces an implicit notion of time, and we introduce two new operators (that resemble the $[\_]\_$ operator in [FM91] for instance):

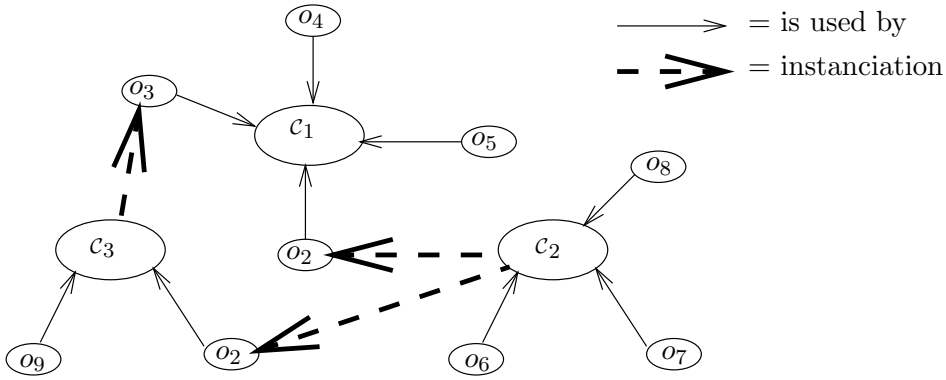
- **after**$\Big[t_1 \,;\, \cdots \,;\, t_n\Big]\Big(\,\varphi\,\Big)$ means intuitively that the formula φ must be true immediately after the term $(t_1 \,;\, \cdots \,;\, t_n)$ has been performed

- **when**$\Big[\varphi_1\Big]\Big(\,\varphi_2\,\Big)$ means that at each time φ_1 would be satisfied, the formula φ_2 must be satisfied.

Finally, an *object type specification* is a couple $SP = (\Theta, Ax)$ where Θ is an object type signature and Ax is a set of well-formed formulas on Θ.

3.2 System specifications

A system is obtained by gluing together several object types. Since object types can be pictured as stars whose branches are partial views of other object types, a system can be naturally pictured as a set of actualization arrows between stars. Arrows go from the center of a star (≈ the actual parameter) to the branch of another one (≈ the virtual parameter).



There are of course some "consistency constraints" in order to ensure compatibility between the assumed partial view in a branch and the exhaustive specification of the corresponding center. For example, with respect to the picture above, it is required that the set of methods M^{o_3} from the star of c_1 (the middle star) is included in $M^{\mathcal{C}_3}$ (from the left star), and there are similar requirements for each instanciation arrow (and on classical data types).

Moreover, to define a *system specification*, we allow "global formulas" in order to express system properties which cannot be specified inside a unique object type. These properties can involve all the symbols from all the stars of the system under consideration.

The semantics of a system specification is defined as a set of models (one model of each object type specification belonging to the system), satisfying some consistency constraints, and satisfying the global formulas.

4 Mapping GNOME into ÉTOILE

In order to give an algebraic semantics to GNOME, we simply chose to provide an algorithm that extract a ÉTOILE-specification from a GNOME specification. Then, the algebraic semantics of GNOME is defined as the set of models of the translated ÉTOILE-specification, under a few restrictions as seen in the next sub-section.

4.1 Main difficulties

In addition to the fact that GNOME is a concrete specification language while ÉTOILE is only a specification formalism, another feature that makes the translation algorithm a little bit complex is that in GNOME, once a method has been `call`ed, its result can be used several times without executing the method again, whilst on the contrary, a ÉTOILE method is executed every time it is referred to in a term. Consequently, it has been necessary to "capture" the result of every called method in a fresh variable before using it in a ÉTOILE term. This gives rise to tedious **when** statements and variable replacements in expressions.

Other minor difficulties have been:

- All `act`ions and `slots` are made visible in the ÉTOILE-specification, and thus we have to keep in mind elsewhere that induction proofs have only to be done with respect to the methods which translate *available* `ser`vices.

- In GNOME, there is no difference between a boolean value and a formula. To handle this peculiarity, we had to provide a boolean function to simulate each predicate. For sake of clarity, this aspect will not be addressed in the rest of the article.

- GNOME has an object identity generator, `next`, a built-in operation which returns a new identity each time it is called. In ÉTOILE, we had consequently to specify a fictitious object *gensym* (generator of symbols) with, for each object type o, a method $next_o :\rightarrow o$, and with axioms saying that $next_o$ never generates twice the same identity, and always succeeds. For lake of space, this aspect will not be addressed in the rest of the article.

4.2 The translation algorithm

A GNOME system specification being given, the translation is simply obtained by translating each GNOME class specification of the system into one object type specification in ÉTOILE.

To get an object type ÉTOILE-specification, we have first to get a ÉTOILE-set (Definition 3.1): the type of interest c is simply the identifier which follows the keyword `class` in the GNOME specification, and for each type name that appears in the GNOME specification, we put it in O if it is specified in another GNOME class specification, else in D.

In addition, GNOME allows methods to return more than one result (the `res` keyword) while ÉTOILE does not. Thus, for every method that has more than one `res`ult, we introduce a fictitious Cartesian product type in D, whose components are the `res` types.

Next, to get a signature (Definition 3.2), we have to fill in the set of functions F:

- for each built-in GNOME data type in D, put the corresponding functions

- for each fictitious Cartesian product type added to D, from a GNOME declaration of the form "`serv` $\cdots$ `res` $name_1 : \varsigma_1$ `res` $name_2 : \varsigma_2 \cdots$ `res` $name_n : \varsigma_n$", we have to add the corresponding tuple generator "$(\_, \_, \cdots, \_) : \varsigma_1\ \varsigma_2 \cdots \varsigma_n \rightarrow prod$" as well as the projections "$name_i : prod \rightarrow \varsigma_i$"

- moreover, GNOME has a special polymorphic constant to address undefined identities, `nil`, thus, for each object type $o \in O$, we put "$nil_o :\rightarrow o$" in F (only if it is used in the specification)

To complete the signature, we have to fill in all the M^o for $o \in \{c\} \cup O$:

- The `ser`vices and the `act`ions are put in $M^{\mathcal{C}}$, and their arity is obtained by collecting all the types of the `par`ameters as domain and taking as codomain: nothing if there is no `res`ult, the type of the result if there is only one, or else, the Cartesian *prod*uct mentioned before.

- For each non derived slot in the `body`, "`slot` $sl : \varsigma$", we define two methods in $M^{\mathcal{C}}$: "$sl :\rightarrow \varsigma$" and "$sl Modif : \varsigma \rightarrow$" (the first one to observe the slot value and the second one to modify it). If the slot is a derived slot, we only define the observer sl.

- Lastly, for each service which is `called` in the body, put it in the corresponding M^o. Its arity is obtained by looking at the GNOME specification of the `class` o, as previously.

To obtain the axioms of the translated ÉTOILE-specification:

- For each fictitious Cartesian product type added to D, and each projection $name_i$ in F, we put the axiom $\quad name_i(x_1, x_2, \cdots, x_n) = x_i$
- For each $nil_o \in F$ we put the axiom $\quad \neg\, \mathbf{alive}(nil_o)$
- For each birth service declared in the GNOME interface,

  $$\texttt{birth serv } s \texttt{ par } p_1 : \varsigma_1 \texttt{ par } p_2 : \varsigma_2 \ \cdots$$

  we put the axiom $\quad \mathbf{after}\Big[(x.s(p_1, p_2 \cdots))\Big]\Big(\mathbf{alive}(x)\Big)$ where the p_i become ÉTOILE variables of sort ς_i and x is a fresh variable of type c.

  Similarly, for each `death service`: $\quad \mathbf{after}\Big[(x.s(p_1, p_2 \cdots))\Big]\Big(\neg\, \mathbf{alive}(x)\Big)$
- For each non derived slot sl and each other slot sl' we put the four following axioms:

  $\mathbf{succeed}(sl\mathit{Modif}(y))$
  $\mathbf{when}\Big[sl' = x\Big]\Big(\mathbf{after}\Big[sl\mathit{Modif}(y)\Big]\Big(sl' = x\Big)\Big)$
  $\mathbf{after}\Big[sl\mathit{Modif}(x)\Big]\Big(sl = x\Big)$
  $sl_{\downarrow} = \texttt{-}_{\downarrow}$

  (a slot modifier always succeeds and only modifies the slot itself).
- For each declared derived slot, "`der slot` $sl : \varsigma$ `by` *expression*", we put the two axioms:

  $sl = expression$
  $sl_{\downarrow} = \texttt{-}_{\downarrow}$

Indeed, the actually deep considerations begin with the translation of the `serv`ices in the `body` of the GNOME specification.

A service is specified in GNOME by a set of alternatives where anyone of them can be executed provided that it is successful, and if no one is successful, the service is not successful[2]:

$$\texttt{serv } s \quad \texttt{alt } \alpha_1 \cdots \quad \texttt{alt } \alpha_2 \cdots \quad \cdots$$

This is translated as a disjunction of "guarded" formulas. The "guard" is a formula stating if the alternative succeeds. Thus, for each service, we write an axiom of the form:

$$\left(\bigvee_{\{\alpha_i\}} \mathbf{when}\Big[\psi_i\Big]\Big(guard_i \wedge \Phi_i\Big)\right) \ \vee \ \left(\left(\bigwedge_{\{\alpha_i\}} \mathbf{when}\Big[\psi_i\Big]\Big(\neg guard_i\Big)\right) \ \wedge \ \mathbf{wait}(s(\vec{x}))\right)$$

The reader should intuitively understand ψ_i as a sequence of "`let` $\cdots$ `in` $\cdots$" statements (like in [AZ92]) in order to prepare the environment of the formula $guard_i \wedge \Phi_i$.
$\vec{x}$, ψ_i, $guard_i$ and Φ_i are explained below.

[2] If there is only one alternative, `alt` α_1 is left implicit

If the action s under consideration is not in the interface, then it has no `par`ameter, thus $\vec{x}$ is empty. Else, in the interface where the service is defined, we collect all the "`par` $p_j : \varsigma_j$" and we let $\vec{x} = (p_1, \cdots, p_m)$, where the parameter names p_j are now considered as ÉTOILE variables of type ς_j.

An `alt`ernative is of the form

$$\texttt{alt}\ \alpha_i \quad \texttt{enb}\ \varphi_i \quad \textit{rest of the alternative}$$

ψ_i is a conjunction of atoms, and $guard_i$ is of the form "$\varphi_i \ \wedge\ \mathbf{succeed}(\tau_i)$", where τ_i is a sequence term. τ_i and ψ_i are built as follows from the *rest of the alternative*:

For each call

$$\texttt{call}\ c\ [\texttt{as}\ c']\ \texttt{of}\ t\ \ \texttt{arg}\ a_1 = t_1 \cdots \texttt{arg}\ a_k = t_k$$

if `as` c' is missing, then choose a fresh variable c' and consider that it is present.

- if t is `next` then add "$y = next_o$" (where y is a fresh variable) to ψ_i and let t' be the term y, else let t' be t itself
- let t'_j be the term t_j where the arguments $s.p_1 \cdots s.p_m$ of the service under consideration are replaced by the variables $p_1 \cdots p_m$ of $\vec{x}$, and where `nil` is replaced by the corresponding nil_o, if applicable
- if c has at least one `res`ult, then add "$\cdots \wedge\ (c' = (t'.c(t'_1, \cdots, t'_k)))$" to the conjunction ψ_i, where c' is now considered as a (fresh) variable
- add "$\cdots$ `;` $(t'.c(t'_1, \cdots, t'_k))$" to the sequence term τ_i

$\mathbf{succeed}(\tau_i)$ means of course that the alternative is successful.

Lastly, Φ_i is the formula "$(s(\vec{x})_{\downarrow} = \tau'_{i\downarrow})$" if s does not return any `res`ult, else Φ_i is the formula "$(s(\vec{x})_{\downarrow} = \tau'_{i\downarrow}) \ \wedge\ (s(\vec{x}) = \vec{r_i})$", where

- τ'_i is the sequence term obtained from τ_i by adding on the right, for each GNOME line "`val` sl `<<` $expr$", the term "$\cdots$ `;` $slModif(expr')$". The expression $expr'$ is obtained from $expr$ as follows:
  - Every result of a call $(c'.name)$ is replaced by, either c' itself if $name$ is the only `res`ult of the corresponding c, or $name(c')$ if c has several `res`ults. (Remember that the ÉTOILE variable c' has been bound in ψ_i, and that, if c has several results, the type of c' is a fictitious Cartesian product and $name$ is a projection.)
  - The `arg`uments $s.p_1 \cdots s.p_m$ of the service under consideration, are replaced by the variables $p_1 \cdots p_m$ of $\vec{x}$.
  - Every occurrence of `next` is replaced by the corresponding ÉTOILE variable y (the one bound in ψ_i).
  - Every occurrence of `nil` is replaced by the corresponding well typed nil_o.
- For each GNOME line of the form "`ret` $s.name_j = expr_j$" in the alternative α_i, let $expr'_j$ be obtained from $expr_j$ as above. If s returns several `res`ults then $\vec{r_i}$ is the vector $(expr'_1, expr'_2, \cdots)$ where the $expr'_j$ are ordered according to the fictitious Cartesian product defined for s, and if s returns only one result, $\vec{r_i} = expr'_1$ (since there is only one `ret` line in this case).

4.3 Example

When we apply this translation to the ProducerConsumer GNOME `specification` given in Example 2.0.1, we have to translate four `class` specifications: ProdConsum, Buffer, Producer and Consumer. The translation of ProdConsum is given as example below.

The signature is defined by:

- $c = ProdConsum$
  $O = \{Producer, Consumer, Buffer\}$
  $D = \emptyset$
- $F = \emptyset$
- $M^{ProdConsum} = \{create :\rightarrow$
  $produce :\rightarrow$
  $consume :\rightarrow$
  $prod :\rightarrow Producer$
  $prodModif : Producer \rightarrow$
  $cons :\rightarrow Consumer$
  $consModif : Consumer \rightarrow$
  $buf :\rightarrow Buffer$
  $bufModif : Buffer \rightarrow\}$
- $M^{Producer} = \{new : Buffer \rightarrow$
  $produce :\rightarrow\}$
- $M^{Consumer} = \{new : Buffer \rightarrow$
  $consume :\rightarrow\}$
- $M^{Buffer} = \{new :\rightarrow\}$

and the axioms are:

- $\mathbf{after}\Big[(x.create)\Big]\Big(\mathbf{alive}(x)\Big)$
- $\mathbf{succeed}(prodModif(y))$
  and 2 other similar axioms for *consModif* and *bufModif* respectively
- $\mathbf{when}\Big[cons = x\Big]\Big(\mathbf{after}\Big[prodModif(y)\Big]\Big(cons = x\Big)\Big)$
  $\mathbf{when}\Big[buf = x\Big]\Big(\mathbf{after}\Big[prodModif(y)\Big]\Big(buf = x\Big)\Big)$
  ... and 2 other similar axioms for *consModif*, and 2 other ones for *bufModif*
- $\mathbf{after}\Big[prodModif(x)\Big]\Big(prod = x\Big)$
  and 2 other similar axioms for *consModif* and *bufModif* respectively
- $prod_{\downarrow} = \text{-}_{\downarrow}$
  and 2 other similar axioms for *cons* and *buf* respectively

- $$\mathbf{when}\Big[(y_1 = next_{Buffer}) \wedge (y_2 = next_{Producer}) \wedge (y_3 = next_{Consumer})\Big] \left(\begin{array}{l} \mathbf{succeed}((y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1))) \\ \wedge \\ create_{\downarrow} = \left(\begin{array}{l} (y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1))\,; \\ bufModif(y_1)\,;prodModif(y_2)\,;consModif(y_3) \end{array} \right)\downarrow \end{array} \right)$$
  $$\bigvee \left(\begin{array}{l} \mathbf{when}\Big[(y_1 = next_{Buffer}) \wedge (y_2 = next_{Producer}) \wedge (y_3 = next_{Consumer})\Big] \\ \qquad \Big(\neg\,\mathbf{succeed}((y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1)))\Big) \\ \wedge\ \ \mathbf{wait}(create) \end{array} \right)$$
  Let us remark that this formula can be simplified as follows:
  $$\mathbf{when}\Big[(y_1 = next_{Buffer}) \wedge (y_2 = next_{Producer}) \wedge (y_3 = next_{Consumer})\Big] \left(\begin{array}{l} \left(\begin{array}{l} \mathbf{succeed}((y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1))) \\ \wedge \\ create_{\downarrow} = \left(\begin{array}{l} (y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1))\,; \\ bufModif(y_1)\,;prodModif(y_2)\,;consModif(y_3) \end{array} \right)\downarrow \end{array} \right) \\ \bigvee \\ \left(\begin{array}{l} \neg\,\mathbf{succeed}((y_1.new)\,;(y_2.new(y_1))\,;(y_3.new(y_1))) \\ \wedge\ \ \mathbf{wait}(create) \end{array} \right) \end{array} \right)$$
  Moreover, *create* being an atomic method, either $\mathbf{succeed}(create)$ or $\mathbf{wait}(create)$ is satisfied; and the satisfaction of an equality ensures that the left hand side succeeds if and only if the right hand side does. Consequently, since slot modifiers are always successful, the previous formula can again be simplified as:
  $$\mathbf{when}\begin{bmatrix} (y_1 = next_{Buffer}) \\ \wedge (y_2 = next_{Producer}) \\ \wedge (y_3 = next_{Consumer}) \end{bmatrix} \left(create_{\downarrow} = \left(\begin{array}{l} (y_1.new)\,; \\ (y_2.new(y_1))\,; \\ (y_3.new(y_1))\,; \\ bufModif(y_1)\,; \\ prodModif(y_2)\,; \\ consModif(y_3) \end{array} \right)\downarrow \right)$$
- $$\mathbf{when}\Big[true\Big]\Big(\mathbf{succeed}((prod.produce))\ \wedge\ produce_{\downarrow} = (prod.produce)_{\downarrow}\Big)$$
  $$\bigvee\ \Big(\mathbf{when}\Big[true\Big]\Big(\neg\,\mathbf{succeed}((prod.produce))\Big) \wedge \mathbf{wait}(produce)\Big)$$
  Similarly, this formula can be simplified as: $produce_{\downarrow} = (prod.produce)_{\downarrow}$
- $$\mathbf{when}\Big[true\Big]\Big(\mathbf{succeed}((cons.consume))\ \wedge\ consume_{\downarrow} = (cons.consume)_{\downarrow}\Big)$$
  $$\bigvee\ \Big(\mathbf{when}\Big[true\Big]\Big(\neg\,\mathbf{succeed}((cons.consume))\Big) \wedge \mathbf{wait}(consume)\Big)$$
  Similarly, this formula can be simplified as: $consume_{\downarrow} = (cons.consume)_{\downarrow}$

5 Proving properties

Let us consider the Producer-Consumer system specification described in Section 4.3, and let us prove the following formula φ for all objects *pc* of type *ProdConsum*:

$$(\varphi) \qquad ((pc.prod).nbprod) = ((pc.cons).nbcons) + ((pc.buf).size)$$

The inference rules we use come from the ÉTOILE-calculus [Aig95b]. They will be introduced here on a "call-by-need" basis, just before their first use.

We firstly prove the following lemma:

> *For every "user-method" $m : s_1 \cdots s_n \rightarrow [s]$ belonging to the Producer-Consumer system (i.e., belonging to the root specification), for every identity variable id of sort ProdConsum, and for every variables x_i of sort s_i, we have:*
>
> $$\varphi \Longrightarrow \mathbf{after}\Big[(id.m(x_1, \cdots, x_n))\Big]\Big(\varphi\Big)$$

Sketch of the proof: For lack of space, we will outline the proof for only one of these methods (*create*). The other proofs are similar. We have to prove:

$$\varphi \Longrightarrow \mathbf{after}\Big[(id.create)\Big]\Big(\varphi\Big)$$

Let us assume that $id = pc$; in this case we will directly prove $\mathbf{after}\Big[(pc.create)\Big]\Big(\varphi\Big)$ without using the precondition φ. The proof can be done via 9 main steps:

1. From the producer specification, we have:
   - $new(x)_{\downarrow} = (mybufModif(x)\,;nbprodModif(0))_{\downarrow}$
   - $\mathbf{after}\Big[nbprodModif(0)\Big]\Big(nbprod = 0\Big)$
2. From the first step, we can write:
   - $\mathbf{after}\Big[mybufModif(x)\,;nbprodModif(0)\Big]\Big(nbprod = 0\Big)$

   the rule we use to obtain the formula above is often called "modal generalization:"

   $$\frac{\mathbf{after}\Big[t_1\,;\cdots;t_n\Big]\Big(\varphi\Big)}{\mathbf{after}\Big[t_0\,;t_1\,;\cdots;t_n\Big]\Big(\varphi\Big)}$$

3. From the steps above we can deduce
   - $\mathbf{after}\Big[new(x)\Big]\Big(nbprod = 0\Big)$

   because the "side effects" of $new(x)$ and $(\,mybufModif(x)\,;nbprodModif(0)\,)$ on the producer state under consideration (*self*) are identical, as expressed by the axiom:
   $new(x)_{\downarrow} = (mybufModif(x)\,;nbprodModif(0))_{\downarrow}$.
4. Since the formula obtained at the third step is satisfied for the representative producer *self*, it is satisfied for every producer y. Consequently, we can write:
   - $\mathbf{after}\Big[(y.new(x))\Big]\Big((y.nbprod) = 0\Big)$
5. Following similar proofs for consumers and for buffers, we obtain:
   - $\mathbf{after}\Big[(z.new(x))\Big]\Big((z.nbcons) = 0\Big)$

- $\mathbf{after}\Big[(x.new)\Big]\Big((x.size)=0\Big)$

6. The 6 following formulas can also be proved without difficulties (x being of sort *Buffer*, y of sort *Producer* and z of sort *Consumer*):
   - $\mathbf{when}\Big[(x.size)=n\Big]\Big(\mathbf{after}\Big[(y.new(x))\Big]\Big((x.size)=n\Big)\Big)$
   - $\mathbf{when}\Big[(x.size)=n\Big]\Big(\mathbf{after}\Big[(z.new(x))\Big]\Big((x.size)=n\Big)\Big)$
   - $\mathbf{when}\Big[(y.nbprod)=n\Big]\Big(\mathbf{after}\Big[(z.new(x))\Big]\Big((y.nbprod)=n\Big)\Big)$
   - $\mathbf{when}\Big[(x.size)=n\Big]\left(\mathbf{after}\begin{bmatrix}bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((x.size)=n\Big)\right)$
   - $\mathbf{when}\Big[(y.nbprod)=n\Big]\left(\mathbf{after}\begin{bmatrix}bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((y.nbprod)=n\Big)\right)$
   - $\mathbf{when}\Big[(y.nbcons)=n\Big]\left(\mathbf{after}\begin{bmatrix}bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((y.nbcons)=n\Big)\right)$

   Consequently, it comes (from 4 and 5):
   - $\mathbf{after}\begin{bmatrix}(x.new)\,;\\ (y.new(x))\,;\\ (z.new(x))\,;\\ bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((x.size)=0\Big)$
   - $\mathbf{after}\begin{bmatrix}(y.new(x))\,;\\ (z.new(x))\,;\\ bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((y.nbprod)=0\Big)$
   - $\mathbf{after}\begin{bmatrix}(z.new(x))\,;\\ bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((z.nbcons)=0\Big)$

7. Using "modal generalization" on the three last formulas, we get respectively:
   - $\mathbf{after}\begin{bmatrix}(x.new)\,;\\ (y.new(x))\,;\\ (z.new(x))\,;\\ bufModif(x)\,;\\ prodModif(y)\,;\\ consModif(z)\end{bmatrix}\Big((x.size)=0\Big)$

- **after** $\begin{bmatrix}(x.new)\,; \\ (y.new(x))\,; \\ (z.new(x))\,; \\ bufModif(x)\,; \\ prodModif(y)\,; \\ consModif(z)\end{bmatrix}\Big((y.nbprod) = 0\Big)$
- **after** $\begin{bmatrix}(x.new)\,; \\ (y.new(x))\,; \\ (z.new(x))\,; \\ bufModif(x)\,; \\ prodModif(y)\,; \\ consModif(z)\end{bmatrix}\Big((z.nbcons) = 0\Big)$

Thus from trivial arithmetic properties:

- **after** $\begin{bmatrix}(x.new)\,; \\ (y.new(x))\,; \\ (z.new(x))\,; \\ bufModif(x)\,; \\ prodModif(y)\,; \\ consModif(z)\end{bmatrix}\Big((y.nbprod) = (z.nbcons) + (x.size)\Big)$

8. Using the following producer-consumer specification axiom:

- **when** $\begin{bmatrix}(y_1 = next_{Buffer}) \\ \wedge (y_2 = next_{Producer}) \\ \wedge (y_3 = next_{Consumer})\end{bmatrix}\left(create_{\downarrow} = \left(\begin{array}{l}(y_1.new)\,; \\ (y_2.new(y_1))\,; \\ (y_3.new(y_1))\,; \\ bufModif(y_1)\,; \\ prodModif(y_2)\,; \\ consModif(y_3)\end{array}\right)\downarrow\right)$

we have:

- **when** $\Big[y_1 = next_{Buffer} \ \wedge \ y_2 = next_{Producer} \ \wedge \ y_3 = next_{Consumer}\Big]$ $\Big(\textbf{after}\Big[create\Big]\Big((y_2.nbprod) = (y_3.nbcons) + (y_1.size)\Big)\Big)$

and from the axioms translating slots properties, it comes

- **when** $\begin{bmatrix}y_1 = next_{Buffer} \\ \wedge\ y_2 = next_{Producer} \\ \wedge\ y_3 = next_{Consumer}\end{bmatrix}\Big(\textbf{after}\Big[create\Big]\Big(buf = y_1 \wedge prod = y_2 \wedge cons = y_3\Big)\Big)$

Since the two formulas above are satisfied for the representative *ProdConsum* object *self*, they are in particular satisfied for *pc*. Consequently, we deduce:

- **when** $\Big[y_1 = next_{Buffer} \ \wedge \ y_2 = next_{Producer} \ \wedge \ y_3 = next_{Consumer}\Big]$ $\begin{pmatrix}\textbf{after}\Big[(pc.create)\Big] \\ \Big(((pc.prod).nbprod) = ((pc.cons).nbcons) + ((pc.buf).size)\Big)\end{pmatrix}$

and since y_1, y_2 and y_3 do not appear in the **when** consequence, we deduce:

- $\mathbf{after}\Big[(pc.create)\Big]\Big(\varphi\Big)$

This proves the lemma when $id = pc$.

When $id \neq pc$, the three following formulas are not difficult to prove (similarly to step 6):

- $((pc.prod).nbprod) = n \Longrightarrow \mathbf{after}\Big[(id.create)\Big]\Big(((pc.prod).nbprod) = n\Big)$
- $((pc.cons).nbcons) = n \Longrightarrow \mathbf{after}\Big[(id.create)\Big]\Big(((pc.cons).nbcons) = n\Big)$
- $((pc.buf).size) = n \Longrightarrow \mathbf{after}\Big[(id.create)\Big]\Big(((pc.buf).size) = n\Big)$

and they trivially imply that when $id \neq pc$:

- $\varphi \Longrightarrow \mathbf{after}\Big[(id.create)\Big]\Big(\varphi\Big)$

and the proof of the lemma is achieved by simple case reasoning.

In addition, the formula φ is obviously satisfied in the empty state (i.e., at the beginning of the system, when no object exists). Moreover, our lemma implies that φ is an invariant whatever a user does with the system. Consequently, φ is proved.

6 Concluding remarks

As announced in the introduction, we have described an algorithm to translate GNOME specifications into ÉTOILE ones. Several primitives have not yet been considered, such as visibility mechanism and event broadcasting.

We never aimed in this article at comparing the already existing alternative semantics of GNOME with the ÉTOILE one, but we shall do so in the near future. Indeed, our main motivation for the moment is to allow GNOME to take benefit of the abstract implementation approach developed in ÉTOILE. And of course, in that respect the ability to reasonably perform proofs was a important and useful indication.

Acknowledgments

The authors are gratful to their colleagues in the IS-CORE and COMPASS projects for many rewarding discussions, and specially to Cristina Sernadas for valuable comments and suggestions.

References

[AB95] M. Aiguier and G. Bernot. Algebraic semantics of object type specifications. In Information Systems Correctness and Reusability, Selected papers from the IS-CORE workshop, September 1994, World Scientific Publishing, R.J.Wieringa R.B.Feenstra eds, 1995.

[ABBI94] M. Aiguier, J. Benzakki, G. Bernot, and M. Israel. Ecos: from formal specification to hardware/software partitioning. Proc. of the VHDL Forum'94, Grenoble, France, 1994.

[Aig95a] M. Aiguier. Spécifications algébriques par objets : une proposition de formalisme et ses applications à l'implantation abstraite. Thèse de Doctorat, janvier 1995, Université de Paris-Sud, Orsay, 1995.

[Aig95b] M. Aiguier. Un calcul pour les spécifications algébriques à objets: l'etoile-calcul. LaMI report 14-95, Université d'Evry, 1995.

[AZ92] E. Astesiano and E. Zucca. A semantics model for dynamic systems. Fourth International Workshop on Foundations of Models and Languages for Data and Objects, Volske, Germany, October 1992, (U.W. Lipeck, B. Thalheim eds.), Springer-Verlag, p.63-80, 1992.

[BEPP87] E.K. Blum, H. Ehrig, and F. Parisi-Presicce. Algebraic specification of modules and their basic interconnections. Journal of Computer Systems Science, Vol.34, p.293-339, 1987.

[CP94] P. Carmo and P. Penedo. GNOME compiler. Research report, Instituto Superior Técnico, Secção de Ciência da Computação, Departamento de Matemática, 1096 Lisboa, Portugal, 1994.

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. OBLOG users manual. Technical report, INESC, Rua Alves Redol 9,1000 Lisboa, Portugal, 1989.

[Dau92] P. Dauchy. Développement et exploitation d'une spécification algébrique du logiciel embarqué d'un métro. Thèse de Doctorat, Juillet 1992, Université de Paris-Sud, Orsay, 1992.

[DG94] P. Dauchy and M.C. Gaudel. Algebraic specifications with implicit state. LRI Report 887, University of Paris-Sud, Orsay, 1994.

[EDS93] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing systems as object communities. Proc. of TAPSOFT'93 (FASE), Orsay, Springer-Verlag LNCS 668, p.453-468, 1993.

[EGR94] H. Ehrig and M. Grosse-Rhode. Functorial theory of parameterized specifications in a general specification framework. Theoretical Computer Science (TCS), Vol.135, p.221-266, Elsevier Science Pub. B.V. (North-Holland), 1994.

[EGS91] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their specification. 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop, Dourdan (France), September 1991, Recent Trends in Data Type Specification, Springer-Verlag LNCS 655 (M. Bidoit, C. Choppy eds.), p.40-65, 1991.

[EM85] H. Ehrig and B. Mahr. Fundamentals of algebraic specification 1. equations and initial semantics. EATCS Monographs on Theoretical Computer Science, Vol.6, Springer-Verlag, 1985.

[ESD93] ESDI, Av. Alvares Cabral 41, 7o, 1200 Lisboa. OBLOG *users manual*, 1993. Supplied with the OBLOG-CASE V1.0 product kit.

[FCSM91] J. Fiadeiro, J.F Costa, A. Sernadas, and T. Maibaum. Objects semantics of temporal logic specification. 8th Workshop on Specification of Abstract Data Types joint with the 3rd COMPASS Workshop, Dourdan (France), September 1991, Recent Trends in Data Type Specification, Springer-Verlag LNCS 655 (M. Bidoit, C. Choppy eds.), p.236-253, 1991.

[FM91] J. Fiadeiro and T. Maibaum. Towards object calculi. Selected papers of the ISCORE'91 workshop, September 1991, London, (G. Saake and A. Sernadas eds.), 1991.

[GD92] J.A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. Recent Trends in Data Type Specification, 5th Workshop on Specifications of Abstract Data Types joint with 4th COMPASS Workshop, Caldes de Malavella, Spain, October 1992, p.1-29, 1992.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. Current Trends in Programming Methodology, R.T. Yeh ed., Printice-Hall, Vol.IV, p.80-149, 1978.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. *Object-oriented specification of information systems: The* TROLL *language.* Technical University of Braunschweig, 1991.

[JSHS95] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – a language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 1995. In print.

[JSS91] R. Jungclaus, G. Saake, and C. Sernadas. Formal specification of object systems. In S. Abramsky and T. Maibaum, editors, *Proc. TAPSOFT'91 Vol.2*, pages 60–82, Brighton, 1991. LNCS 494, Springer-Verlag.

[Jun93] R. Jungclaus. *Modeling of Dynamic Object Systems.* Vieweg Verlag, Wiesbaden/Braunschweig, 1993.

[NOS95] M. Navarro, F. Orejas, and A. Sanchez. On the correctness of modular systems. Theoretical Computer Science (TCS), Vol.140, p.139-177, Elsevier Science Pub. B.V. (North-Holland), 1995.

[Ram95] J. Ramos. Lógica de certificação GNOME. Master's thesis, Instituto Superior Técnico, 1995.

[SCS92] A. Sernadas, J.-F. Costa, and C. Sernadas. Especificação de objectos com diagramas: Abordagem OBLOG. Research report, Instituto Superior Técnico, Secção de Ciência da Computação, Departamento de Matemática,1096 Lisboa, Portugal, 1992. Prémio Descartes 1992, in portuguese.

[SGG+92] C. Sernadas, P. Gouveia, J. Gouveia, A. Sernadas, and P. Resende. The reification dimension in object-oriented database design. In D. Harper and M. Norrie, editors, *Specification of Data Base Systems*, pages 275–299. Springer-Verlag, 1992.

[SGS92] C. Sernadas, P. Gouveia, and A. Sernadas. OBLOG: Object-oriented, logic-based conceptual modelling. Research report, Instituto Superior Técnico, Secção de Ciência da Computação, Departamento de Matemática,1096 Lisboa, Portugal, 1992.

[SR94] A. Sernadas and J. Ramos. A linguagem GNOME: Sintaxe, semântica e cálculo. Research report, Instituto Superior Técnico, Secção de Ciência da Computação, Departamento de Matemática,1096 Lisboa, Portugal, 1994.

[SRGS91] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large object-oriented design of information systems. In F. van Assche, B. Moulin, and C. Rolland, editors, *The Object-Oriented Approach in Information Systems*, pages 209–232. North-Holland, 1991.

[SS94] A. Sernadas and C. Sernadas. Object certification. In A. Olivé, editor, *Fifth International Workshop on the Deductive Approach to Information Systems*, pages 55–78. UP Catalunha, 1994. Full version submitted for publication in the journal Knowledge and Data.

[SSC95] A. Sernadas, C. Sernadas, and J. F. Costa. Object specification logic. *Journal of Logic and Computation*, 5, 1995. In print. Earlier versions available as research reports since 1992.

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

[WM95] M. Walicki and S. Meldal. Multialgebras, power algebras and complete calculi of identities and inclusions. Recent Trends in Data Type Specification, Springer-Verlag LNCS 906, 1995.

A Implemented buffers in ÉTOILE

This appendix presents an abstract implementation of buffers as linked lists of cells.

The signature **Cell** is defined by:

- $c = Cell$
  $O = \{Elem\}$
  $D = \emptyset$
- $F = \emptyset$
- $M^{Cell} = \{new : Elem\ Cell \rightarrow$
  $value :\rightarrow Elem$
  $next :\rightarrow Cell$
  $nextModif : Cell \rightarrow\}$
- $M^{Elem} = \emptyset$

and the axioms are:

- $\mathbf{after}\Big[(x.new(e,c))\Big]\Big(\mathbf{alive}(x) \wedge (x.value) = e \wedge (x.next) = c\Big)$

- $value_{\downarrow} = \text{-}_{\downarrow}$
- $\mathbf{after}\Big[nextModif(c)\Big]\Big(next = c\Big)$
- $next_{\downarrow} = \text{-}_{\downarrow}$

The signature **Implemented Buffer** is defined by:

- $c = ImplBuffer$
  $O = \{Cell, Elem\}$
  $D = \emptyset$
- $F = \emptyset$
- $M^{ImplBuffer} = \{new :\rightarrow$
  $put : Elem \rightarrow Elem$
  $get :\rightarrow Elem$
  $first :\rightarrow Cell$
  $firstModif : Cell \rightarrow$
  $last :\rightarrow Cell$
  $lastModif : Cell \rightarrow\}$
- $M^{Cell} = \{new : Elem\ Cell \rightarrow$
  $value :\rightarrow Elem$
  $next :\rightarrow Cell$
  $nextModif : Cell \rightarrow\}$
- $M^{Elem} = \emptyset$

and axioms are:

- $\mathbf{after}\Big[(x.new)\Big]\Big(\mathbf{alive}(x)\Big)$
- $\mathbf{after}\Big[firstModif(c)\Big]\Big(first = e\Big)$
- $first_{\downarrow} = \text{-}_{\downarrow}$
- $\mathbf{after}\Big[lastModif(c)\Big]\Big(last = c\Big)$
- $last_{\downarrow} = \text{-}_{\downarrow}$
- $get = (first.val)$
- $get_{\downarrow} = firstModif(first.next)_{\downarrow}$
- $put(e)_{\downarrow} = (new(e, last)\,;(last.setnext(x))\,;setlast(x))_{\downarrow}$

B Stacks in GNOME

In this appendix we present a specification in GNOME of stacks as linked lists of nodes.

```
class Stack
  interface
    birth serv new
    serv push par elm: int
    serv ptop res elm: int
  body
    slot depth:nat
    slot tn:Node
    serv new
      val depth<<0
      val tn<<nil
    serv push
      call new of next:Node
        arg new.nn=tn
        arg new.dt=push.elm
      val depth<<depth+1
      val tn<<Node.next
    serv ptop
      enb not(depth=0)
      call del of tn:Node
      val depth<<depth-1
      val tn<<del.nn
      ret ptop.elm=del.dt
end Stack
```

```
class Node
  interface
    birth serv new par nn:Node par dt:int
    death serv del res nn:Node res dt:int
  body
    slot nn:Node
    slot dt:int
    serv new
      val nn<<new.nn
      val dt<<new.dt
    serv del
      ret del.nn=nn
      ret del.dt=dt
end Node
```

C Slidding Window Protocol in GNOME

In this appendix we present a specification of the slidding window protocol [Tel94]. This protocol allows information to be sent between two processes P and Q, through a *Channel*. The assumptions, requirements and protocol are completely symmetric w.r.t. P and Q. The input of P consists of the information it must send to Q, the *array in*. The output of P consists of the information it receives from Q, and is also modeled by an *array out*. There are constants known by both P and Q, lp and lq. The constant lp (resp. lq) fixes the number of messages that P (resp. Q) can send *ahead* of Q (resp. P), i.e., without having received an acknowledgement from Q (resp. P).

This algorithm has been proved to be correct, i.e., *no message is lost*, *no message is duplicated* and *messages don't change order*, using the temporal semantics defined for GNOME and the associated proof system.

```
class P
  interface
    birth serv new
      par in: array(data) par c: Channel
    serv rec
      par i: int par w: data
  body
    slot s: int
    slot a: int
    slot in: array(data)
    slot out: array(data)
    slot i:int
    slot cha: Channel
    serv new
      val s<<0
      val a<<0
      val in<<new.in
      val out<<undef
      val i<<0
      val cha<<new.c
    serv rec
      alt newmsg
        enb out[rec.i]=undef
        val out[rec.i]=rec.w
        val a<<max{a,rec.i-lq+1}
        val s<<min{j:out[j]=undef}
        val i<<max{i,rec.i-lq+1}
      alt old msg
        enb not(out[rec.i]=undef)
    act send
      enb a<=i and i<s+lp
      call recp of cha: Channel
        arg recp.i=i
        arg recp.w=in[i]
      val i<<if i=s+lp-1 then a else i+1
end P
```

```
class Q
  interface
    birth serv new
      par in: array(data) par c: Channel
    serv rec
      par i: int par w: data
  body
    slot s: int
    slot a: int
    slot in: array(data)
    slot out: array(data)
    slot i:int
    slot cha: Channel
    serv new
      val s<<0
      val a<<0
      val in<<new.in
      val out<<undef
      val i<<0
      val cha<<new.c
    serv rec
      alt newmsg
        enb out[rec.i]=undef
        val out[rec.i]=rec.w
        val a<<max{a,rec.i-lp+1}
        val s<<min{j:out[j]=undef}
        val i<<max{i,rec.i-lp+1}
      alt old msg
        enb not(out[rec.i]=undef)
    act send
      enb a<=i and i<s+lq
      call recq of cha: Channel
        arg recq.i=i
        arg recq.w=in[i]
      val i<<if i=s+lq-1 then a else i+1
end Q
```

```
class Channel
  interface
    birth serv new par p: P par q:Q
    serv recp par i: int par w:data
    serv recq par i: int par w:data
  body
    slot mp:list(pair)
    slot mq:list(pair)
    slot p: P
    slot q: Q
    serv new
      val mp<<newlist
      val mq<<newlist
      val p<<new.p
      val q<<new.q
    serv recp
      alt save
        val mq<<app((recp.i,recp.w),mq)
      alt loose
    serv recq
      alt save
        val mp<<app((recq.i,recq.w),mp)
      alt loose
    act sendp
      enb not(empty(mp))
      call rec of p:P
        arg rec.i=head(mp).i
        arg rec.w=head(mp).w
      val mp<<tail(mp)
    act sendq
      enb not(empty(mq))
      call rec of q:Q
        arg rec.i=head(mq).i
        arg rec.w=head(mq).w
      gval mq<<tail(mq)
end Channel
```

We assume defined a structure data type *pair* with two fields: i of type int and w of type data.