A Brief Introduction to GNOME*

Jaime Ramos†
Amílcar Sernadas‡

September 27, 1995

1 Introduction

GNOME [10] is a simplified and revised version of the object-oriented specification language OBLOG [14, 13, 9, 15, 3, 2]. 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 [7, 5, 4, 6] is another noteworthy dialect of OBLOG.

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 won't 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.

As to datatypes, GNOME has a fixed set of basic datatypes (eg, integers, booleans, lists, etc...), but, in opposition to OBLOG, doesn't have any specific constructors for datatype specification.

A formal semantics for GNOME, based on temporal logic, has been defined in [8] on an enrichment of OSL [12, 11]. 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 [1].

2 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.

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

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

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

Example 2.1 *(Producer-Consumer)* Assume we want to specify a system with a producer, a consumer and a buffer with unbounded capacity.

```
class 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
    slot st:(c,g)
    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
```

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

A class in GNOME has two main components: the `interface` and the `body`. In the interface, we declare the public actions (`serv`ices) 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 atributes of the class (`slot`s), the internal `act`ions and define the methods of all actions. A method can have four types of clauses: `enb`ling 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);
  - evaluate the expressions on the valuation clauses (in the present example, the expression c);
  - assign the result of the evaluations to the corresponding slots;

- 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.

Example 2.2 *(System initialization)* Assume we wanted to specify the initialization of the system. There is GNOME a special class with a single instance, the `root`, that is created at the beginnig of the system, and therefore can be used to specify the startup conditions.

```
root
  body
    on bigbang act startup
      call new of next: Buffer
      call new of next: Producer
        arg new.bf=Buffer.next
      call new of next: Consumer
        arg new.bf=Buffer.next
end root
```

The action *startup* is a reaction to the event `startup`, broadcasted by the system at the begining.

3 Outlook

Current research work on GNOME is aimed at:

- developing and comparing alternative semantics:
  - object-algebras;
  - event-structures;
  - quantales;
  - typed calculus of objects;
  - non-sequential automata;
  - situation calculus;
- further extending the language in order to support feature-oriented specifications;
- clarifying the semantics of overriding and feature introduction within default logic.

A Stacks

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
```

B Slidding Window Protocol

In this appendix we present a specification of the slidding window protocol [16]. This protocol allows information to be sent between two processes P and Q, through a *Channel*. The assumptions, requirements and protocol are completly 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 consistes 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.

References

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

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

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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