Sequential Simulation in Prosit: Programming Model and Implementation

Philippe Mussi, Günther Siegel

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sequential Simulation in Prosit: Programming Model and Implementation

Philippe Mussi, Günther Siegel

N° 2713

Novembre 1995

PROGRAMME 1

ISSN 0249-6399

Sequential Simulation in Prosit: Programming Model and Implementation

Philippe Mussi, Günther Siegel*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Sloop

Rapport de recherche n° 2713 — Novembre 1995 — 26 pages

Abstract: The Sloop project, launched in 1995, works on three main domains: (1) discrete event simulation, (2) parallel object-oriented languages and (3) interconnection networks. One of the research objective of the project is the development of Prosit, a sequential and distributed object-oriented workbench for discrete event simulation. Object orientation is used to ensure two main concerns:

- performance on conventional, networked and multi-processor machines
- versatility and ease of use.

We have now achieved the first part of Prosit. We have completely specified the simulation and the execution models, and we have a sequential C++ version of the simulator.

This report concentrates on the modelling process and the object oriented architecture of the sequential simulator. As an illustration of possible applications, we present the classes supporting the queueing network paradigm.

Key-words: Discrete event simulation, Object oriented simulation, Queueing network, C++

(Résumé : tsvp)

A short version of this report has been presented at the 7th European Simulation Symposium (Erlangen, Germany, October 26-28, 1995).

*{Philippe.Mussi},{Gunther.Siegel}@sophia.inria.fr

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Simulation Séquentielle en Prosit : Modèle de Programmation et Implémentation

Résumé : Le projet Sloop, débuté en 1995, porte sur l'étude et le développement des méthodes et des outils permettant l'utilisation efficace de machines multi-processeurs pour la simulation de systèmes à événements discrets. Un des objectifs du projet est le développement de Prosit, un nouvel environnement de programmation orienté objet pour la simulation à événements discrets. L'utilisation du paradigme objet devant permettre :

- d'obtenir de bonnes performances à la fois sur les architectures mono-processeur, multi-processeurs et multi-processeurs virtuels.
- une grande souplesse et facilité d'utilisation.

La première étape de Prosit est maintenant achevée. Nous avons spécifié le modèle de simulation ainsi que le modèle d'exécution, et nous avons une implémentation séquentielle du simulateur.

Ce rapport détaille le processus de modélisation et présente l'architecture orientée objet développée. Comme exemple d'utilisation, les classes supportant la simulation de réseaux de files d'attente sont introduites.

Mots-clé : Simulation à événements discrets, Simulation orientée objet, Réseau de files d'attente, C++

Contents

1 Introduction

The SLOOP project, launched in 1995, works on three main domains: (1) discrete event simulation, (2) parallel object-oriented languages and (3) interconnection networks. One of the research objective of the project is the development of PROSIT, a sequential and distributed workbench for discrete event simulation. Its design is based on the object paradigm and its main objective is to allow *target independence*: distributed (in both optimistic and conservative variants) and sequential simulation without user code modification.

The design philosophy and the modelling process in PROSIT have been presented in [MM93], an overview of the implementation strategies was given in [FMMS94].

We have now achieved the first part of PROSIT. We have completely specified the simulation and the execution models, and we have a sequential C++ [ES90] version of the simulator[1]. This sequential prototype will allow us to investigate the usability and power of expression of our modelling and programming process. Furthermore, it will be used in order to demonstrate the feasability of *target independence* and to measure the acceleration obtained when going from sequential to distributed.

This report concentrates on the object oriented architecture of the simulator. As an illustration of possible applications, we present the classes supporting the queueing network paradigm.

The rest of this report is organized as follows. In Section 2, the modelling process is presented. Section 3 describes the queueing network library, and Section 4 details the architecture and implementation of a sequential PROSIT environment. Final comments, conclusions and future investigations are given in Section 5. We give in Appendix A a short simulation example.

2 Modelling process

2.1 An object oriented concurrent model

A PROSIT simulation can be thought of as a collection of concurrently active objects interacting, via service calls, in the simulated time. An active object encapsulates both the specification and the behavior of the modelized entity.

2.1.1 Active simulation objects

Active simulation objects are the basic components that make up the model[2]. An active object executes its *main activity* in an autonomous way, independently of, and concurrently with, other active objects. Active objects can also have *secondary activities*. The activities are all running concurrently in the simulated time[3].

[1] Two students from ESSI (Ecole Supérieure en Sciences Informatiques): L. Bollini and L. Pigallio, have worked during their final year project on the first PROSIT prototype.

[2] e.g., in a queueing network model, queues and customers are active objects.

[3] In this sequential version all the activities are running pseudo-concurrently in the real time.

When created (using standard C++ creation rules), the active object is automatically managed by the kernel, and is ready to be activated. Its activation time is indicated at creation as a parameter of the constructor. When activated, the object begins to execute its *main activity*.

During its *life* (Fig 1), the object can either be in a running mode (currently executing or consuming time), in a blocked mode (the *main activity* is blocked due to a synchronous request) or in a sleeping mode (it has put itself in *idle–wait state*, waiting to be re–activated by another object). The object is considered to be dead[4] when its *main activity* has terminated. We distinguish normal death (normal termination of the *main activity*), suicide (the object decided to terminate prematurely) and assassination (another object has used the termination primitive).

All the secondary activities of a sleeping active object are also suspended, and they are automatically destroyed when the object dies.

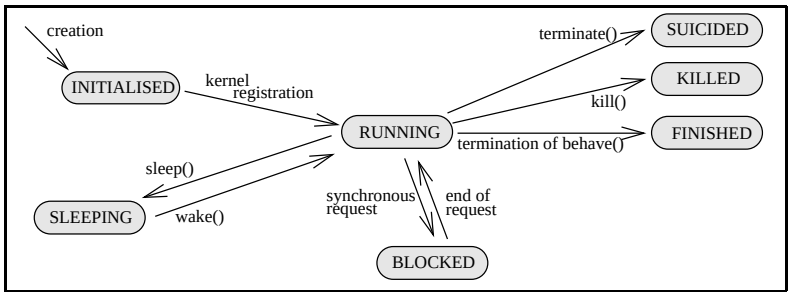

Figure 1: Possible states for an active object.

To be active, an object must be instantciated from a class (*active class*) inheriting directly or indirectly from the system class `Active_Obj`. An active class must at least define the main activity of its instances.

2.1.2 Activities

An activity is an action performed by the active object[5] and corresponds to the execution of a member function. The *main activity* executes the `behave()` function and *secondary activities* are attached to other functions.

An activity has a duration in the simulated time and can halt and be later reactivated (Fig 2). Between the suspension and the reactivation some predefined or random time may have elapsed. An activity will halt either when explicitly consuming time (`wait()`) or when making a synchronous request to an active object.

[4] The death of an active object is distinct from its destruction by the runtime (destructor call and recovery of memory space). A dead active object is an active object that is no longer considered, by the kernel, to be reactivable.

[5] e.g., in a queueing network simulation, customer behaviors and service executions are activities.

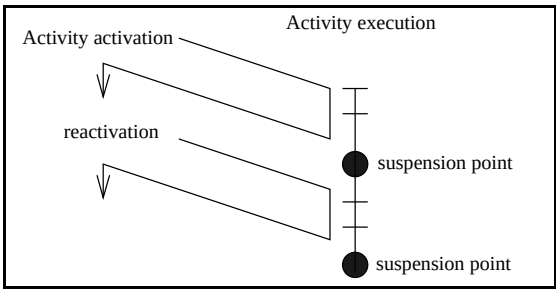

Figure 2: Non-continuous execution of an activity.

An activity terminates when the corresponding function finishes. We have also implemented system primitives allowing an activity to terminate explicitly or to kill an other one. During its execution, an activity goes through different states (Fig 3). An activity is an instance of the system class `Activity`.

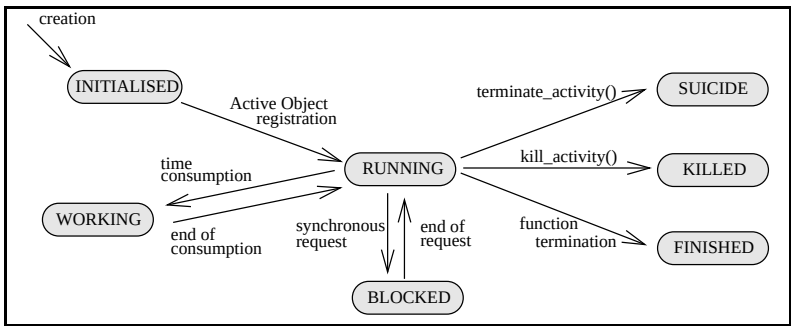

Figure 3: Possible states for an activity.

2.1.3 Interaction

The basic interaction between active objects is the service call, and we have unified member function calls and service calls. A service call can be seen as sending a request to an active object. The call can either be synchronous (the calling activity blocks until termination of the service) or asynchronous (the calling activity carries on immediately). Synchronous calls can have return values.

The request management policy and the service progress depend on the receiving object characteristics. The receiver decides whether, when and how to execute the service. This means that a request can be fully or partly served or even not served at all. In order to take appropriate actions, the caller of a synchronous request not fully served is always informed; in this version of Prosit, an exception is raised in the context of the receiver.

The library has a set of functions that can be used to reject a request or abort a service, and a hierarchy of exception classes for the different cases. This mechanism can easily be extended or modified.

We also distinguish two levels of parameterization in a service call: (1) the service parameters which are related to the model semantics, (2) the control parameters related to the execution, by the active object, of the service (eg. priority, maximum execution time, etc.).

2.2 Building a simulation

The modelling process is an assembly of class instances, using dedicated class libraries and user-defined classes. The PROSIT programmers and advanced-users can develop high-level model libraries. These libraries may include sub-models for high-level subsystems, or highly optimized simulation classes for commonly used sub-systems.

Coding a simulation consists in coding the interfering entities and constructing the model by instantiating and initializing the components.

2.2.1 Component specification

For each different simulated component there must be a class embodying the corresponding functional behavior and implementing the desired services. Three situations can arise:

1. The class already exists in the library.
2. The class is not in the library but a system class with the corresponding functional behavior exists. The simulationist simply has to code a class, inheriting from the system class, and having new members (data and function) implementing the services.
3. The functional behavior of the component is specific, therefore the simulationist has to code it. This can be done in a systematic way thanks to the adopted architecture and the facilities provided by object oriented languages (mainly overloading and redefinition). This new class must define the internal management of arriving requests and the policy for creating and scheduling activities.

2.2.2 Model construction, compilation and execution

Constructing the simulated model consists in creating and initializing an instance of the corresponding class for each interfering entity. By applying the `run()` function to the globally declared kernel, the simulationist starts the simulation. The simulation can be compiled and executed as any C++ program.

2.3 Related works

There are strong links between our active objects approach and the Actor [AH87] and Reactive Objects [BL95] models. Our execution architecture is also greatly inspired by the coroutine facility of Simula [Lam82].

Many C++ packages providing discrete event process based simulation have been proposed: Awesime [Gru91], C++SIM [LM93], SIM*PlusPlus* [Ros92a, Ros92b], SIM++ [BL89], etc.

All of them introduce in the C++ object model the concept of activity. But this extension is performed differently: Awesime, C++SIM and SIM++ introduce active objects; SIM*PlusPlus* has active methods. The multi-active model of PROSIT is unique and can be seen as a combination of both techniques. Further more, PROSIT activities are themselves first class objects (we have a uniform object oriented model).

In the packages supporting the active object model, the simulation entities either interact through events (SIM++), or by using synchronization objets – generally semaphores – (Awesime, C++SIM). The reification[6] of function calls and their unification with service calls are specific to PROSIT . In our environment we provide:

1. for the developer, hierarchical control (encapsulation is respected) and the capacity to program at different levels (new services, new policies, etc.) in a fully object oriented way (everything is an object: activities, requests, etc.);

2. for the simulationist, complete abstraction of the execution features.

[6] Reification transforms a call issued to an object into an object itself.

3 Example of a queueing network library

3.1 Application of the PROSIT model

When applying the PROSIT model to the queueing network paradigm, two kinds of active objects come into sight:

- the customers, who are mono-active objects. Their activity consists mainly in issueing service requests and synchronization operations.
- the stations, who receive requests, from customers or other stations, and process them.

3.1.1 Station specification

The specification of a queueing station is no more different from what has been presented in 2.2.1. System classes supply different combinations of requests (FIFO, LIFO, etc.) and services (Preemption, Processor Sharing, etc.) management policies. Depending on the station, the simulationist will have to supply additional information (number of servers, maximum queue size, etc.).

3.1.2 Customer behavior specification

The coding of a customer behavior is straightforward: the simulationist has to declare a new class, inheriting from the system class `Active_Customer`, and having:

- as data member, typed pointers to the stations at which the customer will ask services.
- as function member, the `body()` function, coding the behavior of the customer as function calls to the data members. Let us note that we have inverted the control of the execution from the server to the customer (we have called this architecture *the customer architecture* [MM93]).

A customer class represents a set of customers whose behavior may be considered as identical. There must be a different class for each *type* of customers.

The simulationist must also specify how the customers will be created in the system by the source stations. A typical source station is characterized by the class of the generated customers and by the intergeneration statistical distribution.

3.1.3 Simulation with passive customers

Sometimes the simulationist will like to describe his model as being a set of servers which must, upon receiving a customer, execute the service and route the customer to the downstream station. In this architecture (we have called it *the server architecture* [MM93]), the customers are passive objects and each station must define the functions `service()`, implementing the service and `route_customer()`, responsible of the routing.

The potential reusability of the classes is reduced but the simulation generally runs faster and consumes less memory. Let us note that both paradigms may be easily mixed, and are supported in PROSIT.

3.2 The developed library

The developed library (Fig 4) has classes for implementing and generating customers, coding service stations, synchronizing activities and programming timers. A simulationist can directly program with these classes, but they may better be used to build higher level and domain specific librairies (e.g. defining a cashier class for a supermarket simulation instead of a FIFO–station class).

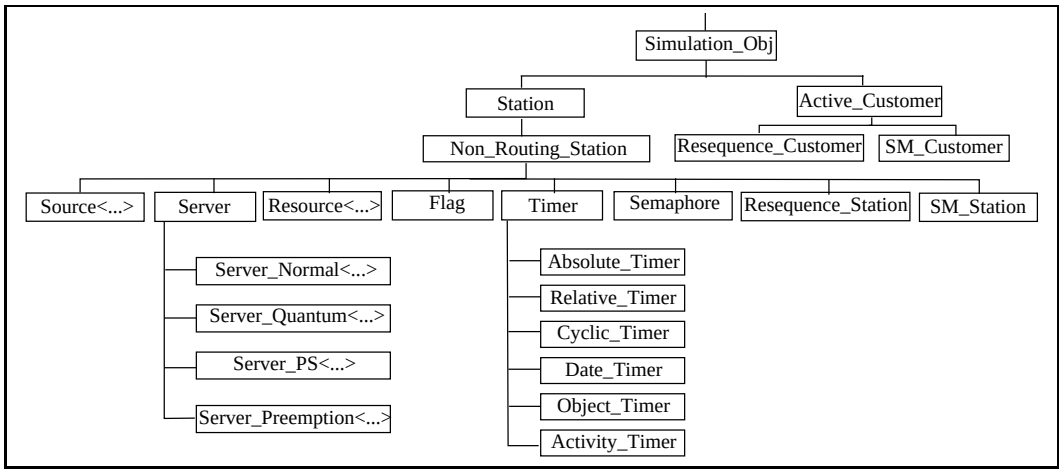

Figure 4: Partial class hierarchy of the queueing network simulation library.

3.2.1 Common functionalities

The following primitives are defined for every simulation entity:

- `get_time()`, which returns the current simulation time.
- `end_simulation()`, to terminate the simulation.

The stations can also overload the boolean function `end_simulation()`. This function is automatically evaluated and returning *true* will trigger the termination of the simulation.

3.2.2 Classes implementing active and passive customers

Passive customers A passive customer class must inherit from the system class `Passive_Customer`.

Active customers An active customer class must inherit from the system class `Active_Customer`. The behavior of the customers must be coded in the `body()` member function.
Two other classes define customers with additional synchronization capabilities:

- `Resequence_Customer`, which must be used if the simulationist wants to re-order the execution of customers. Two primitives are defined: `register_sequence()` and `re_sequence()`.
- `SM_Customer` which must be used if the simulationist wants to split and merge customers. Two primitives are defined: `split()` and `merge()`.

3.2.3 Classes implementing source station

The library provides a generic class (`Source<...>`), parameterized by the generated customers class, and supporting, as a constructor parameter, the statistical distribution object. The simulationist declares a new class inheriting from this system class in order to achieve specific customer initializations. The initialization is performed by the `init_customer()` member function.

3.2.4 Classes implementing service station

For each service scheduling policy, the library provides a generic class (Table 1), the genericity parameter is used to specify the request management policy (Table 2). The programmer can also specify the number of servers (as a parameter to the base constructor) and the maximum queue length (with the `set_queue_size()` primitive).

Class	Scheduling policy
`Server_Normal<...>`	A service is executed without preemption.
`Server_Quantum<...>`	Each service receives cyclically a fixed amount of server time.
`Server_Preemption<...>`	A running service can be preempted by a request having a higher priority level. Two policies are available concerning the continuation of the preempted service: with or without restart.
`Server_PS<...>`	Requests are served immediately, each service receives an amount of server time proportional to the number of running services.

Table 1: Service station classes.

Genericity parameter	Request management policy
FIFO	First come, first served.
LIFO	Last come, last served.
PRIORITY_FIFO	Requests have a priority level, requests having the same level are managed in FIFO order.
PRIORITY_LIFO	Requests have a priority level, requests having the same level are managed in LIFO order.

Table 2: Available request management policy.

3.2.5 Classes implementing synchronization

Different kinds of stations may be used for synchronization (semaphore, flag, resource manager, etc.) and different operations are possible, ranging from the basic ones up to the more sophisticated (customer execution ordering, splitting and merging customers, etc.). The classes are presented in Table 3.

Class	Purpose	Existing Primitives
`Resource<...>`	Manages a set of ressources.	– `take()` – `return()`
`Semaphore`	Low level synchronization based on the classical semaphore concept.	– `p()` – `v()`
`Flag`	Low level synchronization based on the classical flag concept.	– `wait()` – `set()` – `unset()`
`Resequence_Station`	Re-order the execution of customers according to a previous registration. The customers must support re-ordering (see Section 3.2.2).	– `register_sequence()` – `re_sequence()`
`SM_Station`	Split and merge customers. The fission/fusion can be made with or without memory. The customers must support fission/fusion (see Section 3.2.2).	– `code_split()` – `code_merge()`

Table 3: Synchronization classes.

3.2.6 Classes implementing timer

A timer is an object used to perform a defined action (statistics gathering, trace output, etc.) at a precise simulation time, or cyclically. The library provides a set of abstract classes

implementing different firing policies. A new timer class inherits, from one of the system classes, a firing policy and must specify, in the `action()` member function, the action to perform at firing time. Table 4 presents the classes and the firing policy associated.

Class	**Firing policy**
Absolute_Timer	At a given time.
Relative_Timer	After a given amount of time.
Cyclic_Timer	Cyclically, every n units of time.
Object_Timer	After every activation of an active object.
Activity_Timer	After every activation of an activity.
Date_Timer	After every time change.

Table 4: Available timer classes.

4 Architecture and implementation highlights

4.1 Implementing active objects

The activity mechanism has been implemented with a coroutine–like facility developed at the University of Washington [Kep93]. At the lowest level, an activity consists of a stack and the processor context. At the programming level, an activity is an instance of the system class `Activity` (Fig 5). An activity is attached to an active object and is bound to the execution of one of its functions.

```
class Activity : public Temporal_Obj {
protected:
   Active_Obj *user;                    // owner of the activity
   Routine func;                        // bounded function
public:
   ActivityStatus get_status();         // status query primitive
};   // end Activity
```

Figure 5: Partial definition of the `Activity` class.

An active object is an instance of a class inheriting from the system class `Active_Obj` (Fig 6). At instantiation, an activity is automatically created and bound to the execution of the `behave()` function. This *main activity* can create and schedule *secondary activities* to serve the incoming requests. The object (Fig 7) has a local scheduling list and queues for the incoming requests and replies.

```
class Active_Obj : public Temporal_Obj {
protected:
   Activity *main_activity;                          // main activity
   Temporal_List<Activity> *secondary_activities;    // list of secondary activities
   virtual void sleep();                             // idle-wait state primitive
   virtual void terminate();                         // suicide primitive
public:
   virtual void behave() = 0;                        // main activity code
   virtual void kill();                              // assassination primitive
   ActiveObjStatus get_status();                     // status query primitive
};   // end Active_Obj
```

Figure 6: Partial definition of the `Active_Obj` class.

By intensively using the properties of the object oriented paradigm and by designing a good class hierarchy, we have been able to implement active objects in an elegant and natural way which is totally transparent to the simulationist.

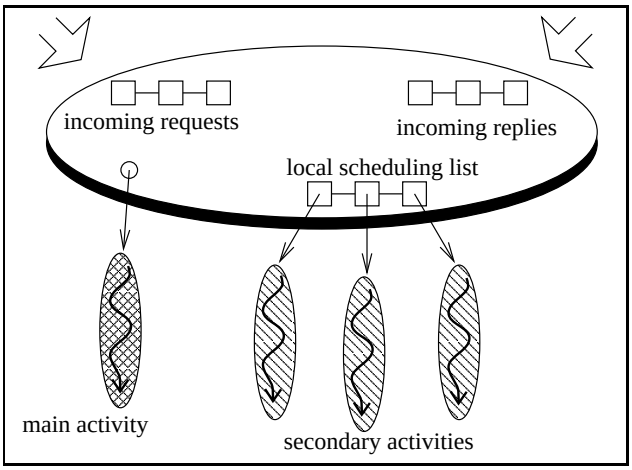

Figure 7: Basic active object architecture

4.2 The service mechanism

From the simulationist point of view, a service call to a station is a function call to the object simulating the station. The PROSIT pre-processor transforms the function call into the construction and emission of a request (Fig 8). The request object holds the service parameters and the control parameters. If the request is a synchronous one, the calling activity will be blocked; if the object is mono-active, the object itself will be blocked.

On the receiving side, the serving object can store the request in a waiting queue or serve it immediately. In order to serve the request, a new activity may be created. For a synchronous request, a reply object will be sent back to the calling object at the end of the service. The reply object holds the termination status, the current simulation date, and the service return values, if any. The reply object will be used to reactivate the calling activity.

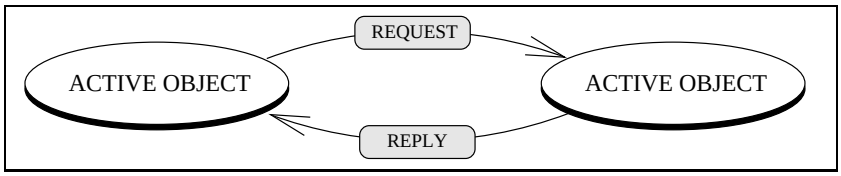

Figure 8: Interaction mechanism

4.2.1 Syntax of a service call

In this version of the simulator we have extended the C++ function call syntax in order to express the specific semantics of a service call (see Section 2.1.3). A service call (see

Fig. 9) is always prefixed by either the `sync` or `async` keyword, indicating the synchronous or asynchronous nature of the call. Following the function call, come the control parameters. The control parameters are directly used to parameterize the request object[7], the system is then fully extensible. See Fig. 10 for a small code example.

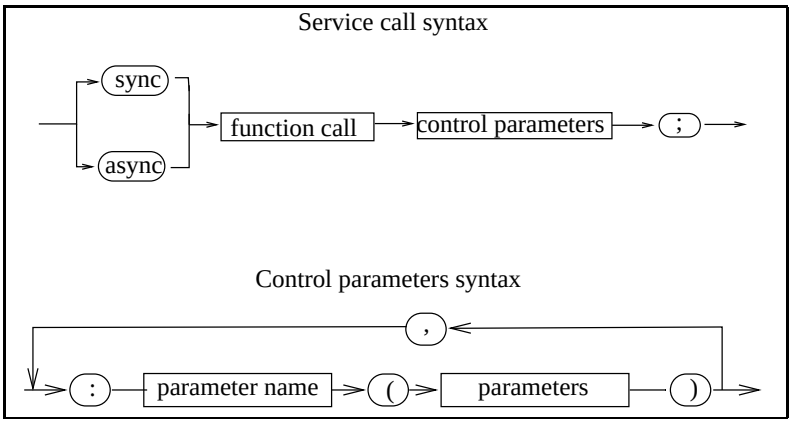

Figure 9: Syntax of a service call.

```
sync station1->service(23) :priority(20), :max_service_time(78);

async station2->another_service();
```

Figure 10: Exemples of service call.

For the exception mechanism used in the request termination status notification (see Section 2.1.3), we use the standard C++ syntax (see Fig. 11). The PROSIT pre-processor can also automatically incorporate in the code the exception handler, but in that case, the associated action will be a default one (ex: suicide of the receiver).

```
try {
   sync station1->service(23) :priority(20);
}
catch (Service_Exception &exception) {
    // code for a non completely served request
}
```

Figure 11: Exception mechanism.

[7] Each control parameter is applied as a member function call on the request object.

4.3 Overall control and time management

4.3.1 Program execution mode

The main PROSIT program supports two different execution modes:

- the *mono-activity* mode, which corresponds to the execution of the `main()` function. The creation and initialization of the model and post-mortem analysis are performed in this mode.
- the *multi-activity* mode, during which the simulation itself runs.

We switch from mono to multi-activity mode when running the kernel; when the simulation finishes, the program returns in mono-activity mode.

4.3.2 Hierarchical control

We have designed a two–level hierarchical control: at the top level, the kernel main loop, in which the kernel manages the reactivation of active objects, and at the object level, the main activity loop, in which the *main activity* manages the reactivation of *secondary activities* (Fig 12).

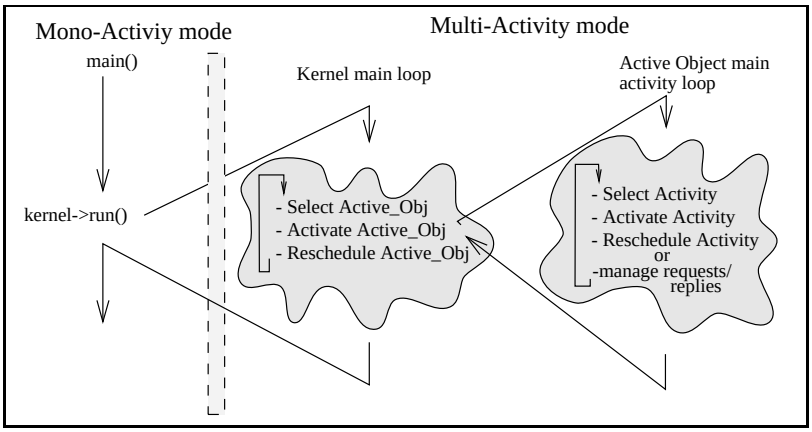

Figure 12: Overall control.

4.3.3 Time management

Each active object or activity has a clock indicating its reactivation time. The kernel extracts, from the scheduling list, the active object with the smallest clock, advances its own clock to the time indicated by the object and reactivates it (in fact it reactivates the *main activity* of the object). If the object is a multi-active one, the *main activity* can either:

- serve waiting requests (creating new *secondary activities*),
- reactivate blocked (if the reply object is arrived) or working (if the requested delay has elapsed) *secondary activities*.

5 Conclusions and future work

During this first step of PROSIT development, we have validated and precisely specified the modelling process for both the final user and the component programmer. We have now a completely specified simulation model and a sequential prototype.

We now focus our investigation in two directions. First, we are working on libraries for different modelling paradigms (Petri nets, task graphs, etc.) and on external components (statistical analysis tool, animation, traces, etc.). Secondly, we are studying the problems arising when distributing the execution of the active objects (modification of the synchronization algorithms, model partitioning, active object migration, load balancing, etc.).

References

[AH87] G. Agha and C. Hewitt. *Concurrent Programming Using Actors.* MIT PRESS, 1987.

[BL89] D. Baezner and EG. Lomow. A Tutorial Introduction to Object-Oriented Simulation and Sim++. In *Winter Simulation Conference*, pages 140–146, 1989.

[BL95] F. Boussinot and C. Laneve. Two Semantics for a Language of Reactive Objects. Technical Report 2511, INRIA, March 1995.

[ES90] M. Ellis and B. Stroustrup. *The Annotates C++ reference manual.* Addison Wesley, 1990.

[FMMS94] P. Ferrante, L. Mallet, P. Mussi, and G. Siegel. Object Oriented Simulation: Highlights on the Prosit Parallel Discrete Event Simulator. In Beaumariage Herring, Wallace and Roberts, editors, *Proceedings of the OO Simulation Conference (OOS'94)*, pages 23–30, Also in INRIA Research Report 2235, 1994.

[Gru91] D. Grunwald. User's Guide to AWESIME-2, A Widely Extensible Simulation Environment. Technical Report CU-CS-552-91, University of Colorado at Boulder, 1991.

[Kep93] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, ftp: ftp.cs.washington.edu, 1993.

[Lam82] G. Lamprecht. *Introduction to Simula 67.* Vieweg, 1982.

[LM93] M. Little and D. McCue. Construction and Use of a Simulation Package in C++. Technical Report 437, University of Newcastle upon Tyne, July 1993. also in: C User's Journal 12(3) March 1994- http://ulgham.ncl.ac.uk/C++SIM/homepage.html.

[MM93] L. Mallet and P. Mussi. Object Oriented Parallel Discrete Event Simulation: The Prosit Approach. In *Modelling and Simulation*, Lyon, June 1993. Also in INRIA Research Report 2232.

[Ros92a] O. Rose. *SIM++ Part I- Sim An Object-Oriented Language for Process-Oriented Discrete-Event Simulation.* Institute of Telematics University of Karlsruhe, 1992.

[Ros92b] O. Rose. *SIM++ Part II- SimEnv An Object-Oriented Simulation Environment for Network Analysis.* Institute of Telematics University of Karlsruhe, 1992.

A Simulation example

In this appendix, we give a short exemple of a supermarket simulation (Fig 13). In Section A.1, the model is coded in the *customer architecture*, and in Section A.2 it is coded in the *server architecture*.

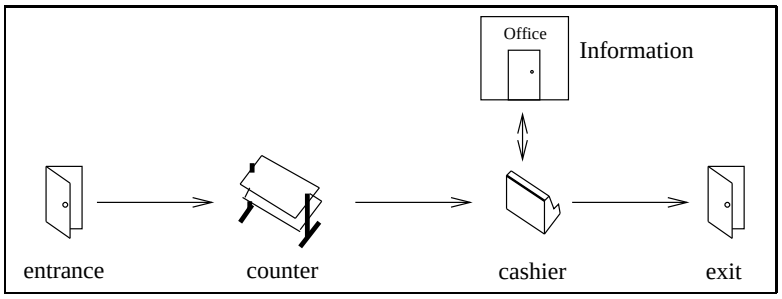

Figure 13: The supermarket model.

A.1 The customer architecture

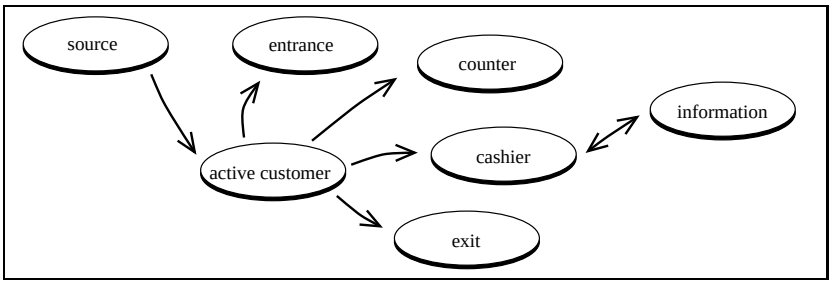

Figure 14: Flow of control in the customer architecture.

The Door class

```
class Door : public Server_Normal<FIFO> {
protected:
    Random *service_distribution;
public:
    void use() { wait((*service_distribution)()); };

    Door(int nb_server,Random *distribution):Server_Normal<FIFO>(nb_server) {
        service_distribution=distribution;
    };
};
```

The Counter class

```
class Counter : public Server_Normal<FIFO> {
protected:
    Random *service_distribution;
public:
    void take(int nb_articles){
        for(int i=1;i<=nb_articles;i++)
            wait((*service_distribution)());
    };

    Counter(Random *distribution) : Server_Normal<FIFO>(100) {
        service_distribution=distribution;
    };
};
```

The Cashier class

```
class Cashier : public Server_Normal<FIFO> {
protected:
    Random *service_distribution, *price_ok;
    Office *superviser;
public:
    void pay(int nb_articles){
         for(int i=1;i<=nb_articles;i++)
             if((*price_ok)())
                wait((*service_distribution)());
             else
                sync superviser->ask_price();
                wait((*service_distribution)());
    };

    Cashier(Office *o,Random *distribution) : Server_Normal<FIFO>(20) {
         service_distribution=distribution;
         superviser=o;
         price_ok=new Binomial(1,0.9, new MLCG);
    };
};
```

The Office class

```
class Office : public Server_Normal<FIFO> {
protected:
    Random *service_distribution;
public:
    void ask_price() { wait((*service_distribution)()); } ;
```

```
    Office(Random *distribution) : Server_Normal<FIFO>(1) {
       service_distribution=distribution;
    };
};
```

The Source class

```
class Main_Source : public Source<Client> {
protected:
    Door *in, *out;
    Counter  *counter;
    Cashier  *cashier;
    Random   *nb_articles_distribution;
public:
    Main_Source((Random *intergeneration, Door *in_, Door *out_, Counter *counter_,
                    Cashier *cashier_) : Source<Client>(intergeneration){
        in=in_;
        out=out_;
        counter=counter_;
        cashier=cashier_;
        nb_articles_distribution=new Uniform(3,20, new MLCG);
    };
protected:
    bool stop_condition() {
        return((total_population()>=1000) || (get_time()>=20000));
    };
    virtual void customer_init(Client *customer)  {
          customer->entrance_door = in;
          customer->exit_door = out;
          customer->visited_counter = counter;
          customer->used_cashier = cashier;
          customer->nb_articles = (int)floor((*nb_articles_distribution)());
    };
};
```

The Customer class

```
class Client : public Active_Customer {
public:
    Door *entrance_door,*exit_door;
    Counter *visited_counter;
    Cashier *used_cashier;
    int nb_articles;
public:
    Client(Time t) : Active_Customer(t) { };
protected:
```

```
    void body(){
      sync entrance_door->use();
      sync visited_counter->take(nb_article);
      sync used_cashier->pay(nb_article);
      sync exit_door->use();
    };
};
```

The main program

```
Kernel *kernel = new Kernel;

main() {

    Office  *superviser_office = new Office(new Uniform(2,4, new MLCG));
    Door    *in                = new Door(20,new Uniform(1,2, new MLCG));
    Counter *counter           = new Counter(new Uniform(1,2, new MLCG));
    Cashier *cashier           = new Cashier(superviser_office,new Uniform(0,1, new MLCG));
    Door   *out                = new Door(20,new Uniform(1,2, new MLCG));
    Main_Source source(new Uniform(0,0.5, new MLCG), in, out,
                           counter,cashier);

    kernel->set_max_simulation_time(50000000);
    kernel->run();
}
```

A.2 The server architecture

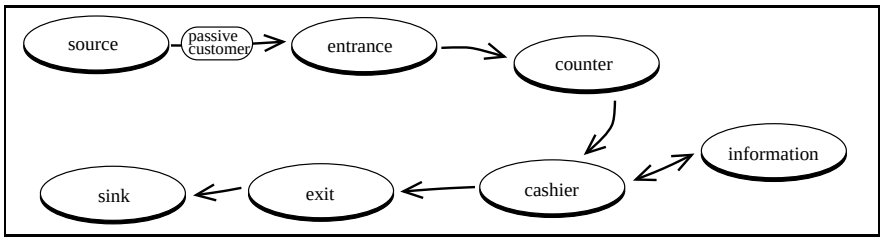

Figure 15: Flow of control in the server architecture.

The Door class

```
class Door : public Routing_Server_Normal<FIFO> {
protected:
    Random *service_distribution;
    Routing_Server  *next;
public:
```

```
    Door(int nb_server,Random *distribution,
              Routing_Server  *next_):Routing_Server_Normal<FIFO>(nb_server) {
      service_distribution=distribution;
      next=next_;
    };
protected:
    void route_customer(Passive_Customer *cl){ async next->receive_customer(cl); };
    void service(Passive_Customer *cl){ wait((*service_distribution)()); };
};
```

The Counter class

```
class Counter : public Routing_Server_Normal<FIFO> {
protected:
    Random *service_distribution;
    Routing_Server  *next;
public:
    Counter(Random *distribution,Routing_Server *next_) :
                                              Routing_Server_Normal<FIFO>(100) {
      service_distribution=distribution;
      next=next_;
    };
  protected:
    void route_customer(Passive_Customer *cl){ async next->receive_customer(cl) };
    void service(Passive_Customer *cl){
      Client *client = dynamic_cast<Client*>(cl);
      for(int i=1;i<=client->nb_articles;i++)
           wait((*service_distribution)());
    }
};
```

The Cashier class

```
class Cashier : public Routing_Server_Normal<FIFO> {
protected:
    Random *service_distribution,price_ok;
    Routing_Server  *next;
    Office *superviser;
  public:
    Cashier(Office *o,Random *distribution,
             Routing_Server *next_):Routing_Server_Normal<FIFO>(20) {
       service_distribution=distribution;
       superviser=o;
       price_ok=new Binomial(1,0.9, new MLCG);
       next=next_;
    };
```

```
  protected:
    void route_customer(Passive_Customer *cl){ async next->receive_customer(cl); };

    void service(Passive_Customer *cl){
        Client *client = dynamic_cast<Client*>(cl);
        for(int i=1;i<=client->nb_articles;i++)
             if((*price_ok)())
                wait((*service_distribution)());
             else
                sync superviser->ask_price();
                wait((*service_distribution)());
    };
};
```

The Office class

```
class Office : public Server_Normal<FIFO> {
protected:
    Random *service_distribution;
public:
    void ask_price(){ wait((*service_distribution)()); };
    Office(Random *distribution):Server_Normal<FIFO>(1) {
        service_distribution=distribution;
    };
};
```

The Terminus class

```
class Terminus : public Sink {
protected:
    void service(Passive_Customer *cl){};
};
```

The Source class

```
class Main_Source : public Routing_Source<Client> {
protected:
    Routing_Server *next;
    Random  *nb_articles_distribution;
protected:
    bool stop_condition(){
        return((total_population()>=1000) || (get_time()>=20000));
    };

    virtual void customer_init(Client *customer) {
       customer->nb_articles = (int) floor ((*nb_articles_distribution)());
```

```
    };

    virtual void route_customer(Passive_Customer *cl) { async next->receive_customer(cl); };
public:
    Main_Source(Random *intergeneration,Routing_Server *next_)
                                        : Routing_Source<Client>(intergeneration){
       next=next_;
       nb_articles_distribution = new Uniform(3,20,new MLCG);
    };

};
```

The customer class

```
class Client : public Passive_Customer {
public:
    int nb_articles;

    Client(Time date): Passive_Customer (date) {};
};
```

The main program

```
Kernel *kernel=new Kernel;
main() {

    Office    *superviser_office = new Office(new Uniform(2,4,new MLCG));
    Terminus  *sink    = new Terminus;
    Door      *out     = new Door(20,new Uniform(1,2, new MLCG),sink);
    Cashier   *cashier = new Cashier(superviser_office,new Uniform(0,1, new MLCG),
                                out);
    Counter   *counter = new Counter(new Uniform(1,2, new MLCG),cashier);
    Door      *in      = new Door(20,new Uniform(1,2, new MLCG),counter);
    Main_Source source(new Uniform(0,0.5, new MLCG), in);


    kernel->set_max_simulation_time(50000000);
    kernel->run();
}
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399