Object oriented simulation : Highlighs on the PROSIT : parallel discrete event simulator

Patrick Ferrante, Philippe Mussi, Günther Siegel, Lionel Mallet

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Object Oriented Simulation:
Highlights on The PROSIT Parallel Discrete Event Simulator

Patrick Ferrante, Philippe Mussi, Günther Siegel, and Lionel Mallet

N° 2235

Avril 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Object Oriented Simulation:
Highlights on The PROSIT
Parallel Discrete Event Simulator

Patrick Ferrante, Philippe Mussi, Günther Siegel [*], and Lionel Mallet [**]

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

Rapport de recherche n˚2235 — Avril 1994 — 24 pages

Abstract: The *Prosit* project, launched in 1991, is devoted to the development of a new discrete event simulation system based on the object paradigm. Object orientation is used to ensure two main concerns:

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

The project described in this paper is a joint effort of Simulog, INRIA and GEC-Alsthom and has been partially supported by French Ministry for Research. A version of this report has been presented at Western Simulation Conference, Tempe (Arizona), January 1994

[*] {Philippe.Mussi},{Gunther.Siegel}@sophia.inria.fr
[**] Simulog S.A., Les Taissounières HB2, Route des Dolines, F-06560 Valbonne(FRANCE), Lionel.Mallet@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

In particular, *Prosit* user programs (models and experiment descriptions) should need only a new compilation so as to run efficiently on either a single mono-processor workstation, a cluster of workstations or a multi-processor machine.

This paper describes the design status of the project. It discusses the implementation strategies for several key aspects of the project. Rather than the model architecture that was presented and discussed in [MM93], it details the *Prosit* simulation environment architecture and addresses many aspects of distributed object oriented programming.

Key-words: Discrete event simulation, Distributed Simulation, Object Oriented Simulation

(Résumé : tsvp)

Simulation orientée objets:
Une introduction au système PROSIT

Résumé : Le projet PROSIT , entamé en 1991, est consacré au développement d'un nouveau système de simulation à événements discrets basé sur le paradigme objets. L'orientation objets a été choisie pour répondre à deux critères principaux:

- la performance, à la fois sur des machines conventionnelles et sur des architectures multi-processeurs
- la versatilité et la facilité d'utilisation.

En particulier, les programmes-utilisateur *Prosit* (modèles et expériences) ne doivent nécessiter qu'une recompilation pour assurer une exécution efficace sur une station de travail classique, un *cluster* de stations de travail ou une machine multi-processeurs.

Ce rapport décrit l'état d'avancement et les choix de conception du projet. Il ne décrit que sommairement l'architecture de modèles (détaillée dans [MM93]), mais détaille l'architecture de l'environnement de simulation et les problèmes de programmation répartie rencontrés.

Mots-clé : Simulation à événements discrets, Simulation répartie, Simulation orientée objets

1 Introduction

It is a well known fact that Discrete Event Simulation and Object Oriented Languages have a long common history. Many authors even consider Simula [Lam88], one of the first simulation languages, as the first real object oriented programming language. However, that common story did not stop back in 1967, as shown by simulation environments like ModSim2 or Sim++ [Ros92].

The authors' organizations have acquired a great experience in discrete event simulation by developing and enhancing the QNAP2[VP85] package, especially in the field of queueing networks. This package has incorporated, over the years, many up to date features, including object oriented enhancements, but its overall architecture made its extension to distributed simulation unlikely.

The *Prosit* project, described in this paper, is devoted to the development of a brand new discrete event simulation system, designed from the grounds up with distributed simulation in mind. Its design is based on the object paradigm, from which naturally derive several interesting features:

1. Modularity and reusability, allowing both *Prosit* programmers and end-users to 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.

2. Target independence: distributed simulation (in both optimistic and conservative variants) and parallel replication is implemented in such a way that application programmers do not have to take it into account. Their simulation classes inherit *parallel methods* only if needed, and the choice of sequential or parallel implementation will only be made at the final compilation stage. Furthermore, simulation classes and user programs are independent of the simulation method used.

3. Extensibility: various tools may be incorporated, with little or no changes in basic or user written classes. These tools include statistics gathering and processing, automatic load-balancing, animation, sub-model aggregation, analytical solvers, etc.

The design philosophy and the modeling process of *Prosit* have been detailed in [MM93]. In this paper, we focus on the implementation strategies used in this

project. We first describe the hardware and software environments on which we build *Prosit* and we briefly explain the changes we had to make to the basic Sather compiler. In section 3, we detail the *Prosit* user interface. Section 4 is devoted to the specific parallel features that we plan to introduce, in order to efficiently implement parallel discrete events simulation in an object oriented environment.

2 Environment

The first *Prosit* prototype is developed in an extended version of the Sather Object Oriented Language [Omo92], and the target system is a Meiko machine.

2.1 Sather

Sather [Omo92] is an object-oriented language designed and implemented at the International Computer Science Institute in Berkeley. It compiles into C and is intended to allow development of object-oriented, reusable software while retaining C's efficiency and portability. It offers different facilities, including a simple and powerful mechanism to interface Sather to C, and an integrated garbage collector.

Our Sather extensions include:

- Cross-compilation, allowing generation of Transputer and Intel i860 executable code from a Sun workstation
- Facilities for parallelism expression and efficient use (transparent calls to remote objects, ability to move objects between processors, parallel virtual memory, etc)
- Light-weight processes and synchronization mechanisms
- A process migration mechanism.

2.2 Meiko

The Meiko machine is a MIMD Computing Surface made of 51 Inmos' Transputers and 16 Intel i860 processors. This system is hosted by a Sun Sparc workstation. The process and inter-process communications are managed by CSN (Computing Surface Network).

3 Simulation Model Compilation and Execution

Let us recall [MM93] that a running simulator is organized in cooperating sub-simulators and a simulation supervisor. Each sub-simulator has a simulation kernel and is in charge of simulating a group of stations of the model. Sub-simulators interact together following a given time management method (e.g., Chandy-Misra, *Time Warp*). The supervisor is in charge of loading the model and of managing the user interface (animation, display of variables, etc) while the simulation is in progress. The supervisor is also used for global communication and other service tasks (debugging, trace facilities, load balancing, etc).

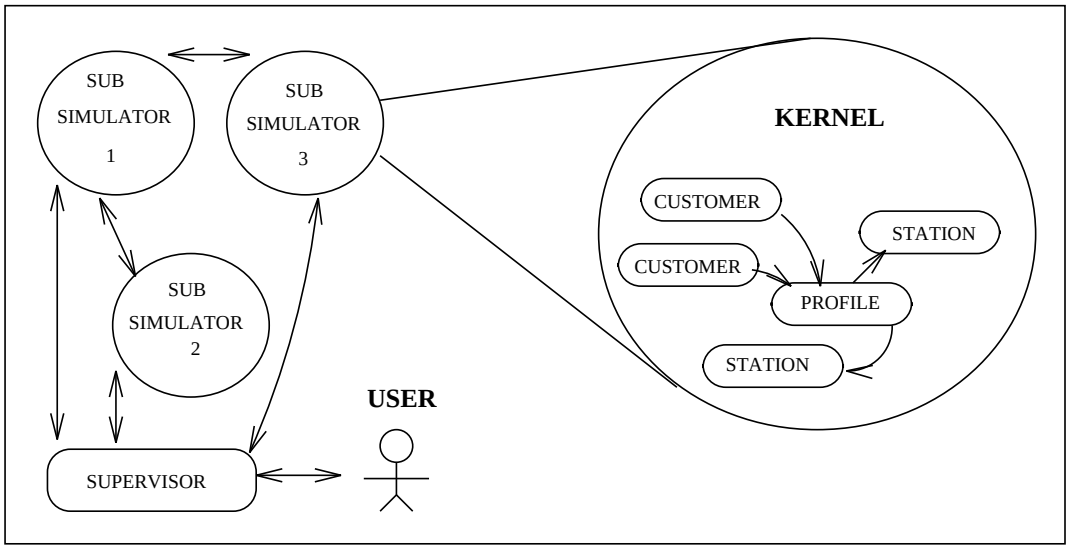

Figure 1: Overall Architecture

3.1 Compilation of the Simulation Model

As described in [MM93], the user builds his model by assembling class instances. But coding a simulation model introduces not only one new dimension, time, but also simulation specific semantics. These constitute the motivation for the *Prosit* language above Sather.

To analyze this language and generate raw Sather code, we can, like many other simulation packages, preprocess the model. But many problems arise from this kind of approach: ambiguous error-messages referring to the preprocessed code which is different from the user code, insufficient compilation checks leading to system-failure at runtime, among others.

Instead, in order to avoid these drawbacks, we have adopted a 2 level compilation:

- The code is pre-compiled by the *Prosit* compiler, which validates the simulation semantics of the program and converts the simulation extensions from the *Prosit* language into raw Sather code.
- The target (Sather) compiler is extended to support parallel primitives. Those primitives are used in an explicit way.

It allows as complete as possible syntactic and semantic validations, from a simulation point of view, of the user code. Furthermore, precompilation allows to hide the distributed aspect of the simulation program, hence providing a higher level of abstraction.

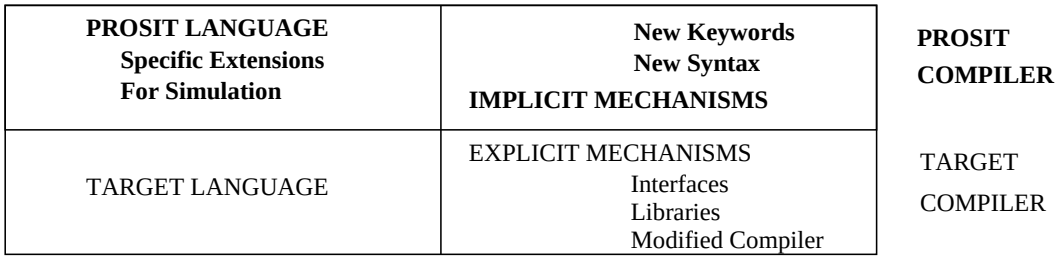

Figure 2: *Prosit* and Target Languages

In the first prototype, both compilers are implemented as separate programs. The *Prosit* compiler generates intermediate files which are processed by the target compiler. In the next generation, both compilers will be merged in order to optimize compilation and offer better error checking.

We propose a single program[1] (SPMD) approach based on a *generic sub-simulator* which can potentially manage all stations. This generic sub-simulator

[1] In an heterogeneous environment, we have one executable for each processor family (our Sather port can generate multiple binaries from a single source code), but in an homogeneous environment we have a single executable.

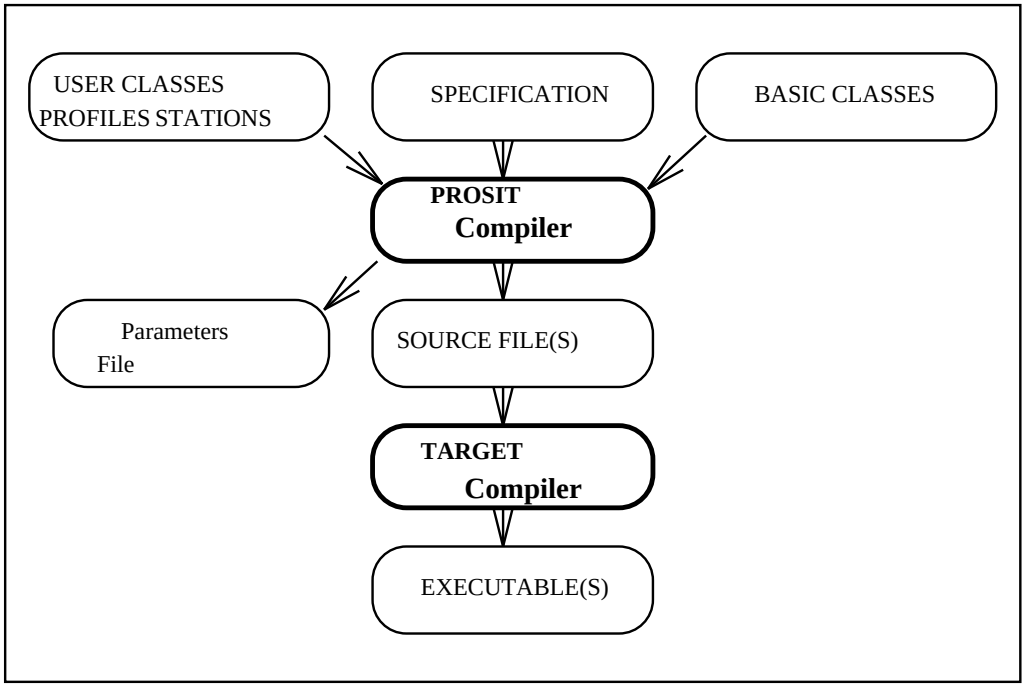

Figure 3: PROSIT Compilation Scheme

is configured at runtime with a list of stations it has to manage. In addition, we are able to dynamically move and create stations. Station migration is an important issue because it allows processor load balancing and decreasing synchronization overheads. Furthermore, having the same classes everywhere is necessary for data exchange and for processes migration.

One can argue that the single executable is bigger than a specific one (containing only the code for the stations managed by the sub-simulator), but in fact the size only depends on the number of station classes, as there is no code duplication for instances in Sather.

During compilation, a configuration file is generated. The simulation supervisor uses this file to know how many sub-simulators to load, on which processing unit, and how to configure them. This information comes from the user code or takes default values. The user can customize the file.

3.2 Execution of the Simulation Program

To execute the simulation, the user runs the supervisor passing the simulation name as a parameter. The supervisor reads the corresponding configuration file and the network resource file (this file lists the processing units available, their architecture, and associated cost values), and then loads and configures as many sub-simulators as specified.

Processing units may be selected according to three different schemes:

- the transparent scheme for which the sub-simulators are automatically assigned to the most appropriate site,
- the *architecture-dependent* scheme for which the user may indicate a specific architecture onto which execute a sub-simulator,
- the *machine-specific* scheme for which the user can choose the machine onto which a sub-simulator will be loaded.

A graphical display system is integrated with the supervisor. It displays on-line figures of *simulation statistics* (server response time, throughput, etc) and *execution statistics* (available memory, processor load, etc). The interface also provides a debug mode allowing to trace the execution at both *user* (service execution, buffer queue, etc) and *system* (communication, processes activities, etc) levels.

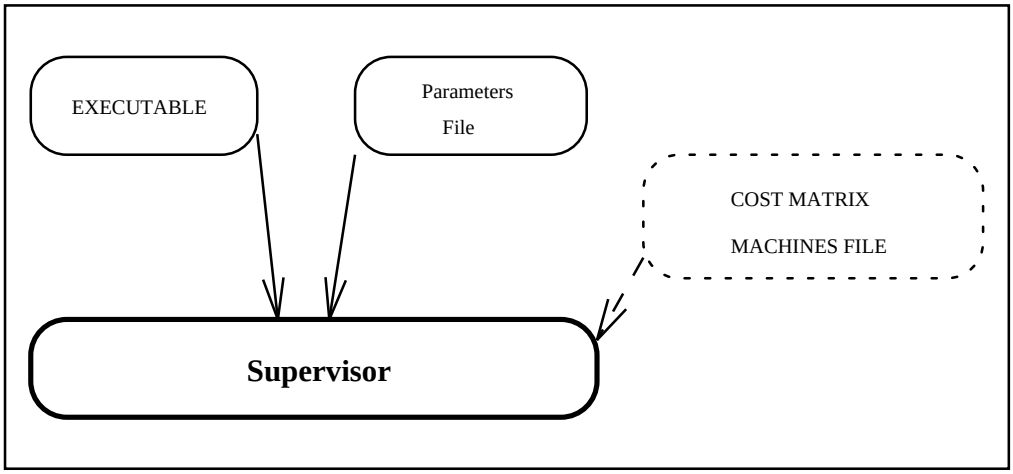

Figure 4: Execution Schemes

4 Parallel Features

4.1 Process Management

Generally, discrete event simulators have to manage the execution of many concurrent entities (stations, sources, etc), and that is particularly true in a **customer architecture** [MM93], where every customer is an active entity. Furthermore, some simulation methods must be *re-entrant* (e.g., multi-server stations methods).

A simple and clear way to achieve the first two constraints (concurrent entities management and re-entrant methods) is to implement entities as light-weight processes. We chose to implement *customers* as processes and *stations* as objects shared by customers. This way, customers concurrency is managed by the underlying operating system, and an execution stack for each customer allows reentrance on station methods. Furthermore, to stick with the **customer architecture** paradigm, as calling a method of a station means moving the customer to the station and requesting the service associated with that method, we need to be able to migrate the associated process to the processing unit where the station is managed. We could also simply use some sort of *Remote Method Call* but we would loose a major benefit of migrating processes: the ability to balance the load of available processing units.

To summarize, *Prosit* simulation environment needs to be able to:

- Create and terminate processes
- Suspend (for a simulation time period) and reactivate the execution of processes
- Synchronize processes
- Migrate processes.

4.1.1 *Prosit* **Processes**

In Object Oriented Languages, processes can be implemented in two ways, either by *process-objects* [Car91] or by *process-methods*. In the process-objects approach, the programmer declares an object as a process by deriving from a special class (e.g., **PROCESS_OBJ**), and runs this process by calling an activation method (e.g., *create*, *live* or *activate*). In the process-methods approach, the programmer can run any method as a process simply by calling the method in a particular way[2]. We choose this solution for two reasons. It allows a more fine-grained parallelism than the process-objects solution, and it is easier to use.

Our implementation introduces three new system classes : the process handler classes PHANDLER0 and PHANDLER, and the GUARD class. Process handlers are used for the process management, and GUARD objects are used for process synchronizations. Every process created by the user may be attached to a process handler. The parameterized PHANDLER must be used to handle process methods which return values. Similarly, PHANDLER0 is designed to handle process methods that do not return values. Process creation is performed by the special assignment operator **:−** which attaches a process handler to the created process. If the programmer is not interested in managing the created process, or retrieving the result, he can create processes using the *anonymous handler identifier* **_**. Otherwise, the process result can be retrieved synchronously with the blocking *take* method of the PHANDLER class.

Processes are created into a system queue (the ACTIVE queue), and are executed in a round robin fashion. The running process can suspend its execution in three ways:

[2]This solution is offered by pSather [FLR92]

```
p : PHANDLER{INT};      -- INT process handler
i : INT;

p :- foo.compute;       -- foo.compute process
_ :- foo.clear_screen;  -- foo.clear_screen anon process
i := p.take;            -- wait for the process result
i := i + 1;             -- now we can use it!
```

Figure 5: Example of Process Use

- by a *schedule* call: the process looses the execution control, and it is inserted at the end of the ACTIVE process queue
- by a *suspend* call: the running process is suspended, it is taken out from the ACTIVE process queue, and stays suspended until another process awakes it by an *activate* call
- by a *wait* call: the running process is suspended until the selected *guard* wakes it up.

The *guard* is a kind of explicit monitor, made of a counter and a process queue. The counter value can be set either at the *guard* creation, or by the *set_counter* method. When a process calls the *wait* method on the *guard*, its execution is suspended, and it is inserted into the *guard* process queue. Each call to the *signal()* method of the *guard*, makes the *guard* counter decrements by one. When the counter reaches 0, all the processes in the *guard* queue are reactivated.

The PHANDLER method *kill* is used to force a process to terminate. The PHANDLER method *current* returns the process handler of the running process. Finally, the PHANDLER method *status* returns the status of a process (ACTIVE, SUSPENDED, GUARDED, FINISHED). The PHANDLER *migrate* method is described later. Figure 7 summarizes the primitives available to the programmer.

This rather simple light–weight processes management system seems to be general enough to implement simulation mechanisms needed by *Prosit* (e.g., the *simulation scheduler* is implemented by a *suspended–processes* queue ordered by processes activation times).

```
g : GUARD;          -- declaration of GUARD
p : PHANDLER0;      -- no-return process handler

g := g.create(1);   -- initialize guard with counter = 1
p :- foo.child(g); -- foo.child process, pass the guard
g.wait;             -- suspend execution until reception
                    -- of 1 signal from child process
foo.compute;        -- proceed
g.set_counter(2);   -- set the guard counter to 2
g.wait;             -- suspend execution until reception
                    -- of 2 signals from child process
```

Figure 6: Example of *guard* use

4.1.2 Process Migration

As said above, a customer should be able to reach remote station services. This can be achieved by either RMC (Remote Method Call), or a *process migration*. As the process migration mechanism is usually a heavy task, it should be used only when it is really useful. Usually, migration is interesting either if the remote method involves heavy computations, or if the same remote method has to be called many times. But it is quite difficult (almost impossible) to foresee the execution of a method. That is why it is difficult to find a good *migration use* strategy.

Our *migration design* takes into account several major constraints: the use of the migration mechanism should be hidden from the user (the simulation model designer doesn't know the actual distribution of simulation entities between sub-simulators) ; the implementation should be as much as possible architecture–independent (only C standard code) and efficient, with an additional cost to the *normal execution* (with no migrations) as low as possible ; and finally, process *migration* should be possible in a heterogeneous environment (different machines with different architectures).

The solution we are currently studying is based on the introduction of the method **migrate(** *destination-sub-simulator* **)** in the PHANDLER Sather system class. A process (e.g., a *customer*) can use this method either on itself, or on another process. The process, on which the method is applied, stops its execution and is inserted into the ACTIVE process queue of the *destination* sub-simulator. As we want to

```
phandler :-  method_call  -- process creation.
```

```
class PHANDLER{RES_TYPE : GENERIC} is
  suspend                  -- process suspension
  schedule                 -- process reschedule
  activate                 -- process reactivation
  is_finished : BOOL       -- process termination test
  take : RES_TYPE          -- return process result
  current : PHANDLER       -- return the running process
  status : STATUS          -- return process status
  kill                     -- kill a process
  migrate(simulator : ID) -- process migration
end

class GUARD is
  wait              -- process suspension on a GUARD
  create(nb)        -- create a new GUARD with counter
  set_counter(nb)   -- set the GUARD counter
  signal            -- send a signal to the GUARD
end
```

Figure 7: Process management primitives

hide parallelism to the model programmer, internal tricks like calls to the **migrate** method should be hidden to him. In order to mask this method call to the model programmer, we are currently studying two strategies based on the *Prosit* compiler.

In the first strategy, the *Prosit* compiler adds in the *customer* code a migration call before each station service method call. This strategy implies a semantic analysis of the whole *Prosit* program in order to detect station methods [3] which are *tagged* as services. Of course, it is not sure that the choice to migrate for every remote service call is always the best choice. Furthermore, with this strategy the **customer** never migrates for *non-service* methods calls, even if it could be convenient.

In the second strategy, the *Prosit* compiler adds into the *customer* code a migration call before each remote station method call. The *simulation classes* programmer is able to optimize the use of the migration mechanism by *tagging* the methods known to be fast. These *tags* are taken into account by the *Prosit* compiler, and *fast* methods are called by RMC.

The main problems we have encountered in our process migration developments were due to the heterogeneous environment :

- basic data types (integers, floats, pointers, etc) could be different in size and in memory representation (*little indians, big indians*, etc)
- stack management (growing stack or decreasing stack).
- some compiler optimizations (local variables allocation into registers instead of the stack, etc).

4.2 Distributed Objects Mechanisms

This section deals with another important issue in distributed object oriented simulation: distributed objects access and management. It details some of the features added to the Sather language and handled by the *Prosit* compiler to hide them to the model programmer. Along this section, the “user” refers to either the user of the extended Sather language or the *Prosit* compiler.

[3] Methods of class which inherit from *STATION* class

4.2.1 The *Remote-Object* model

Our first model for distributed objects management is based on the concept of *remote object* and on multiple inheritance.

A *remote object* is a special entity that is used to access an object, called the *foreign object*, physically located on a remote processor. Methods activation or attributes manipulation, on the *remote object*, are performed through communication. Programs are organized in cooperating sub-tasks (written in the same programming language and having the same classes). A sub-task willing to share an object must export it to the *object manager*, and a sub-task who wants to use a foreign exported object must import it.

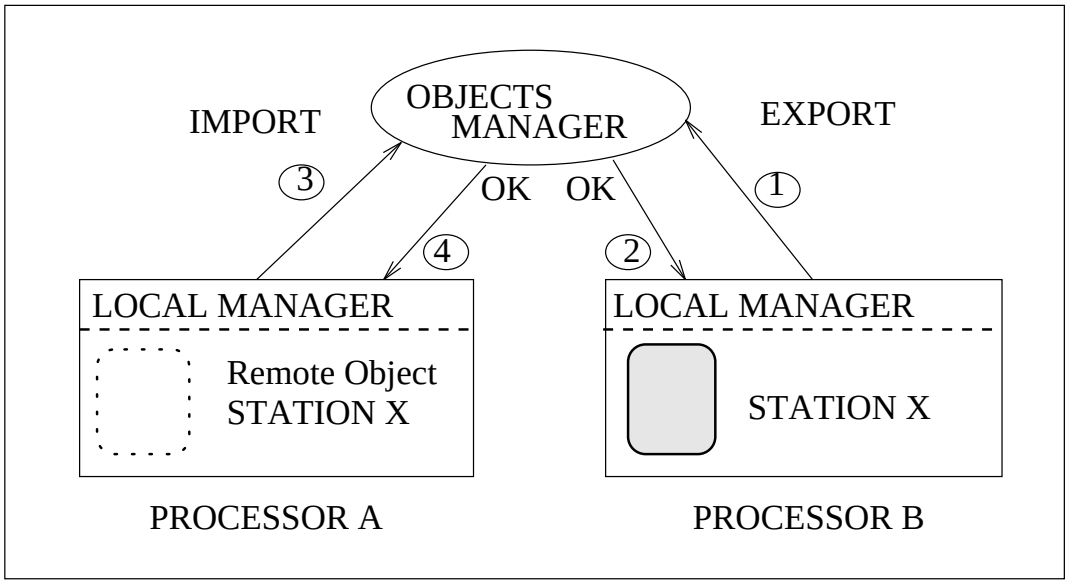

Figure 8: *Export/Import Mechanism*

The object manager is in charge of the import/export operations. When exporting an object, the manager checks that no object with the same *id* has already been exported. At importation, if there is an object corresponding to that *id*, it checks that the types are compatible.

We want to manipulate a *remote object* as a *local* one and we must guarantee that it is associated with a *foreign object* and that the method call or the attribute

manipulation is valid. The semantic check can be performed by the compiler if the *remote* and *foreign* objects have the same interface. The difference is that in the *remote object*, the code is replaced by a communication stub. Those mechanisms are implemented through multiple inheritance.

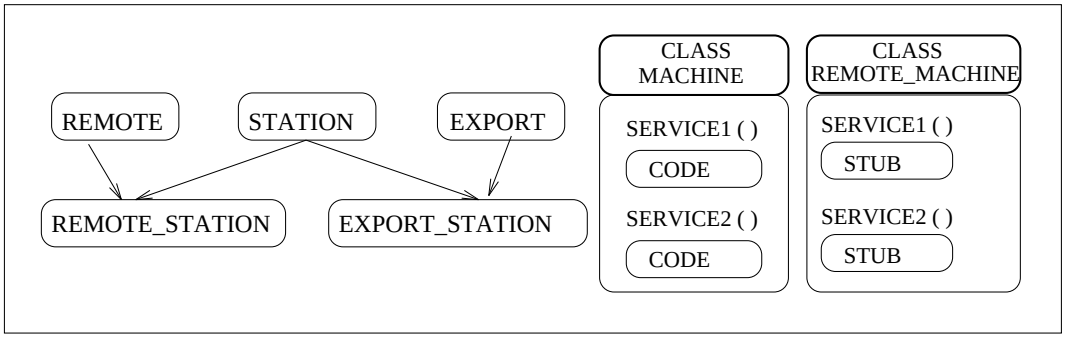

Figure 9: Remote Objects Mechanism Classes

To be exportable an object has to inherit (directly or indirectly) from EXPORT. If the user (in fact the *Prosit* compiler) wants to export an object of class XXX, he will only have to declare the class EXPORT_XXX (subtype of EXPORT and XXX) and declare his object as instances of EXPORT_XXX.

```
class EXPORT is
  registering_status : STATUS  -- object status
  id : IDENTIFICATION          -- object identifier

  register() : BOOL            -- export object
  un_register() : BOOL         -- cancel export
end
```

Figure 10: The EXPORT Class features

In the EXPORT class there is an *id* attribute that stores the object identifier, a *registering_status* attribute that indicates the object's status (*exported*, *not_exported*, *error*), and two methods handling the exportation (*register()*) and un-exportation

(*un_register()*) of the object. Other methods are used only by the runtime environment, in order to handle remote requests.

Similarly, if the user needs a remote object of class XXX, he has to declare the class REMOTE_XXX (heritage of REMOTE and XXX), to create an instance of that class, to initialize it with the *id* of the *remote object*, and finally to call the *import* method on it.

```
class REMOTE is
  registering_status : STATUS  -- object status
  id : IDENTIFICATION          -- object identifier

  import() : BOOL              -- import object
  blocking_import()            -- blocking import
  release() : BOOL             -- unimport object
end
```

Figure 11: The REMOTE Class features

We want to code the operations on *remote object* as *remote procedure call* to the *foreign object*. In the EXPORT_XXX classes, the compiler adds, for each attribute, a *reader* and a *writer* method (cf. CLOS and Sather v1.0).
For the REMOTE_XXX classes, the compiler replaces the XXX method's code with communication stubs and also adds the *reader*/*writer* stubs.

4.2.2 The *virtual memory* model

The *virtual memory* model is much more general and powerful than the previous one. It offers a shared address space between processes running on separate hosts and supports a shared-memory abstraction. The model is based on the pSather [FLM91, FLR92], dpSather [Sch92] programming languages and on the Bird-Meertens operators [Ski90].

The user distributes his program by creating a cluster object on each computer and by starting sub-tasks among those clusters. An object can be created on the local cluster or on a remote one, the runtime environment offers a global memory address space on top of the clusters. Some objects can be replicated on each cluster allowing

data-parallel programming, others can migrate between clusters. All those parallel extensions are introduced through new system classes and through new syntactic constructions.

The cluster The cluster is the abstract entity that the programmer manipulates to represent a processor and its memory. To create a new cluster, on a remote machine, the programmer has to instantiate the system class CLUSTER, with the target computer as a parameter. To specify the target machine, he uses the persistent class[4] HOST which is used to store the network resources. This class offers various selection primitives: specific host, less loaded, etc.

```
Class HOST
  Inherits from : PERSIST_OBJ

  list : HASH{PROC}                -- list of available
                                   -- processors

  add(id : PROC) : BOOL            -- add a processor
  remove(id : PROC) : BOOL         -- remove a processor
  specific(name : STRING) : PROC   -- specific processor
  best() : PROC                    -- less loaded proc.
  any() : PROC                     -- any processor
  arch(type : ARCH_ID) : PROC      -- specific arch.
  test(name : STRING) : C_STATUS   -- test a processor
end
```

Figure 12: The HOST Class

An address in the global memory address space (*long* address) consists of two parts: the cluster number and the object address (*short* address) in the cluster memory.

When an object is created on a cluster, by default it will be on the local cluster, but we can specify the target location (cf. figure 13).

With such a global addressing scheme, remote objects are accessed and used just as if they all were local, thus greatly simplifying remote objects method call.

[4]this class has a single instance loaded at the beginning of the execution

```
i, j : FOO;
.....
i := FOO::new;                 -- local
j := FOO::cluster_new(node1); -- on cluster node1
```

Figure 13: Example of Remote-object Creation

Replicated object A replicated object is an object that is copied on all clusters. A process can read/write its own copy of the local variable or perform combine operations on all the values. The combine operations are based on the Bird-Meertens operators [Ski90] (*map, reduce, prefix*).

The programmer creates replicated classes by inheriting from REPLICATE_OBJ. If an attribute is declared with the *update* qualifier, a modification on a copy is performed on all the replicated variables (by a reliable broadcasting protocol that guarantees that all processes experience changes *in the same order*).

Replication is useful to reduce communication overheads and to more easily program global combine operations (e.g., *scatter, gather*).

Such objects are used for simulation statistical results.

Migrating object Migrating classes are created by inheriting from RELOCATE_OBJ. To migrate an object, the programmer simply calls the *relocate* method on this object, with the target cluster as a parameter.

Migration is useful to balance the load of the different processors from the supervisor. Simulation performances can also be improved this way.

The *virtual memory* model allows better static inconsistency check of the model and provides more flexibility to the model programmer. The communication overhead it implies is roughly balanced by the possibilities it offers to dynamically accommodate the simulation.

4.3 Distributed Garbage Collection

A useful capability offered by the Sather compiler is its integrated *garbage collector* (GC). At the same time it simplifies the dynamic memory management to the user,

and it reduces the *unused-memory* leaks usually induced by an explicit memory deallocation.

Unfortunately, the original Sather GC has been designed for sequential applications, so it is not fully compatible with the *Prosit* distributed environment. In fact it does not take into account references on *remote* objects introduced by both *Prosit* implementation models (*remote objects*, *virtual shared memory*).

This is why we are currently studying different existing distributed garbage collection algorithms [Hug85, DRM93, SDP92] and the modifications they require in order to make a suitable GC for our simulation system.

5 Conclusion

The *Prosit* project, originally devoted to distributed simulation now also addresses very interesting topics in distributed object oriented languages. In fact, distributed object oriented languages probably aren't mature enough to be directly used in such a project. Instead of focusing on the distributed simulation problems (*Time Warp* optimizations, Chandy-Misra improvements and both methods implementations), we also have to concentrate on the language itself. But this has helped us better specify and design the *Prosit* simulator.

Sather seems to be a good choice as it is still evolving and quite *open* to user suggestions. Hopefully and probably because of that choice, we do not spend to much time on implementing the Sather extensions.

We are now expecting very interesting figures from the *Prosit* simulator to help us improve it and determine the real interest and feasability of an industrial distributed object oriented queueing network discrete event simulator.

References

[Car91] Denis Caromel. *Programmation parallèle asynchrone et impérative: études et propositions (Une extension parallèle du langage objet Eiffel)*. PhD thesis, Université de Nancy I Centre de Recherche en Informatique de Nancy (INRIA Lorraine), 1991.

[DRM93] Cédric Dumoulin, Jean-François Roos, and Jean-François Mehaut. Le ramasse-miettes du projet PVC-BOX. In *5èmes rencontres sur le parallélisme*. Laboratoire informatique de Brest, May 1993.

[FLM91] J. Feldman, C. Lim, and F. Mazzanti. Parallel SATHER, language design and application experience. Technical report, International Computer Institute, U.C. Berkeley, 1991.

[FLR92] J. Feldman, C. Lim, and T. Rauber. The shared-memory language pSATHER on a distributed-memory multiprocessor. Technical report, International Computer Institute, U.C. Berkeley, 1992.

[Hug85] John Hughes. A distributed garbage collection algorithm. In *Functional Language an Computer Architectures*, pages 256–272. Jean-Pierre Jouannaud, September 1985.

[Lam88] Guenther Lamprecht. *SIMULA - Einfuehrung in die Programmiersprache*. F. Vieweg und Sohn, Braunschweig ; Wiesbaden, 1988.

[MM93] L. Mallet and P. Mussi. Object oriented parallel discrete event simulation: The prosit approach. In *Modelling and Simulation FSM 93, june 7-9, LYON*. SCS, June 1993.

[Omo92] S. Omohundro. The Sather language. Technical report, International Computer Institute, U.C. Berkeley, 1992.

[Ros92] Ortwin Rose. *SIM++ An object-oriented langage for process-oriented discrete-event simulation User-Manual*. University of Karlsruhe Insitute of Telematics, Febrary 1992.

[Sch92] H. Schmidt. Data-parallel object-oriented programming. In *Fifth Australian Supercomputer Conference*, pages 263–272, 1992.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfosse. SSP CHAINS: Robust, distributed references supporting acyclic garbage collection. Technical Report 1799, INRIA, November 1992.

[Ski90] D. Skillicorn. Architecture-independant parallel computation. *IEEE Computer*, pages 38–50, December 1990.

[VP85] M. Véran and D. Potier. Qnap: a portable environment for queueing systems modelling. In North Holland, editor, *Modelling Techniques and Tools for Performance Analysis*, 1985.

List of Figures

Contents

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de 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