The *SLOOP* project: Simulations, Parallel Object-Oriented Languages, Interconnection Networks

Françoise Baude, Fabrice Belloncle, Jean-Claude Bermond, Denis Caromel, Olivier Dalle, Eric Darrot, Olivier Delmas, Nathalie Furmento, Bruno Gaujal, Philippe Mussi, Stéphane Perennes, Yves Roudier, Günther Siegel, Michel Syska

I3S-CNRS – Univ. de Nice Sophia Antipolis – INRIA
Rte des Colles, B.P. 145,
06903 Sophia Antipolis Cedex, France
{First.Last-Name}@inria.fr

1 Introduction

We give an overview of the *SLOOP* project. A synthetic view of its organization, and each component are presented. This presentation however, primarily focuses on the parallel object-oriented language and simulation aspects.

2 The *SLOOP* project

Our group (INRIA–I3S/CNRS–Univ. of Nice Sophia Antipolis) works in three main areas:

1. parallel and distributed discrete event simulations,
2. parallel object-oriented languages,
3. interconnection networks.

These three different research goals articulate one with another in the construction of the *Sloop* system: each level uses the primitives and possibilities of the layer just underneath; Figure 1 summarizes the system structure and topics.

The bottom layer deals with: communication algorithms, mapping strategies (both static and dynamic), and overlapping communication/computations.

These primitives are used to define and program the middle layer, a parallel object-oriented language (C++//, an extension of C++), which in turn offers: reusability, flexibility, and extensibility in concurrent programming. We achieve these features with polymorphism between objects and processes: a variable statically defined with an object type (not a process) can dynamically reference a process.

Finally, the third level defines in C++// a library of classes to program the model to simulate, which provides a distributed execution of the simulation.

Altogether, *SLOOP* is not a self-contained project since we are using external components in order to program each level. At the bottom, the interconnection network layer uses communication primitives such as PVM [25], and hardware specific primitives for efficiency (shared memory, etc.). The simulations environment layer requires a statistics library in order to analyze results, persistence of data. Finally, the parallel object-oriented language level would benefit from replication for efficiency and possibly for resilience, atomicity, and transactions.

Our system uses such primitives, components, or libraries for its implementation, but in order to be as flexible as possible, we try to give the final user some control over the building blocks, and algorithms being used – in order to map the system on a specific architecture for instance. This trend is sometimes referred to under the name *open implementation* [20].

3 Parallel and Distributed Discrete Event Simulation

The simulation layer offers a set of simulation and modeling C++ classes (simulation classes are those dealing with the simulation phase whereas modeling classes are those used to build the model). To mask the simulation paradigm to the final user of the simulator, a programmer can define a set of classes for a specific field of application. These classes, gathered in libraries, allow to build a model at a higher description level than the one corresponding to the simulation paradigm, eg. by manipulating *cashiers* instead of FIFO

servers in a supermarket simulation. This possibility to define high level libraries is crucial if we want the simulator to be usable by engineers of the simulation domain. In fact, the simulation paradigm itself is contained in the description and code of the simulation classes, and hence hidden. At the end, the model is built by describing a *main* class made of instances of user-defined and system classes.

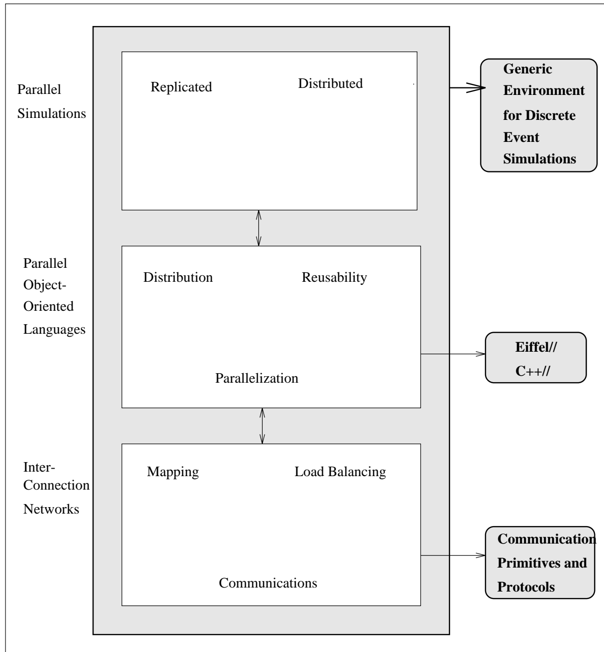

Figure 1: The *SLOOP* project

In other simulation systems, the user describes a simulation model as being a set of servers which (1) execute some kind of service on the customers they receive, and then, (2) forward the customers to another server. Active entities in the model are the servers; they decide what to do with the customer they are processing. The path of customers inside the model can be obtained only by analyzing all server descriptions to retrieve information about customer transit; this structuring can be named the *server architecture*.

In our system, we decided to revert the control of execution from the server to the customer: the customers are active entities in the model, they decide themselves their path of transit in the set of servers. In a more object-oriented fashion, the user programs the customer with a *behavior* method that is activated for each instantiation of a customer of that type; we call this structuring a *customer architecture* by opposition to the usual server-oriented architecture.

For instance, in a supermarket simulation, the objects *entrance, shelving1, ..., shelvingN, exit* being servers, the *behavior* method of a customer (a supermarket client in that case) will be defined as:

```
class Client: public Customer{
...
public:
Void behavior (...) {
   entrance->enter (...);
   shelving1->serve (...)
   ...
   shelvingN->serve (...);
   exit->exiting (...);
}
...
```

We believe this inversion to allow for more reusability. For instance, the servers do not have the path followed by the customers embedded within their code, making both customers and servers classes more self-contained, and thus more reusable. Another important issue in simulation being statistics collection (server, end-to-end, and client statistics), the customer architecture provides the user a frame where client measurement can be placed: the client itself. With the server architecture, these measure points have to be placed into the code of service methods of server objects, which is again incompatible with self-contained clients and servers.

Regarding the parallel and distributed execution, both *Time-Warp* and *Chandy-Misra* time management methods can be used. Parallelism is implemented in a transparent manner for the simulation programmers: their simulation classes do not have to take it into account at the modeling stage. The choice of sequential or parallel execution is only made at a final stage, thanks to some predefined classes which permit to obtain parallel execution when inheriting from them. These predefined classes and parallel behavior are programmed using the parallel object-oriented language described in the next section.

4 Parallel Object-Oriented Language

We defined an extension of C++ called C++//; this work extends the work done with the design of Eiffel// [14]. While Eiffel// offers a specific model of parallel programming, we defined with C++// a first layer which is a set of language primitives, independent of any parallel paradigm, and which permits us to build libraries of nearly all concurrent programming models. Indeed, such language primitives are a Meta-Object Protocol (MOP).

There are various MOPs, with various goals, compile and run-time costs, and various levels of expres-

siveness. Within our context, we propose a reflection mechanism which uses reification. Reification is simply the action of transforming a call issued to an object into an object itself; we say that the call is "reified". From this transformation the call can be manipulated as a first class entity: stored in a data structure, passed as parameter, sent to another process, etc.

A MOP mechanism for C++ being defined, various libraries of concurrent programming models can be designed and implemented. In the remainder of this section, we do not detail the MOP, but we just give an overwiew of the parallel library we use for most of our applications, and especially for programming the simulation classes and their parallel execution.

Processes

This library implements a MIMD model, with sequential processes, asynchronous communications, wait-by-necessity [11] (implicit futures), and without shared objects. It is based on the introduction of a *Process* class.

All objects which are an instance of a class which publicly inherits from the *Process* class are processes (active object = process). Passive objects (ie. objects which are not active) belong at run-time to a process object, thus giving a distributed structure of active objects encapsulating passive objects.

Syntax:

```
class Parallel_A : public A, Process{
...
Parallel_A* p;
p = New Parallel_A ( ... );
```

Communication

A communication between active objects is simply programmed as an asynchronous member function call.

The function calls between processes are implicitly (by default) asynchronous, thus allowing and encouraging parallel execution of objects (although a synchronous method call is also supplied but must be stated explicitly in the function call itself or in the process definition). Function calls between passive objects are implicitly (by default) synchronous as happens in standard sequential C++.

Syntax:

```
p -> f ( parameters );
```

Synchronization

A simple rule permits to deal with asynchronous function calls: *wait-by-necessity*. This mechanism is an implicit (user transparent) future mechanism.

When starting an asynchronous function call, the caller does not wait for the return value until it is explicitly used for some computation. Should the value not have been returned at this point, a wait is automatically triggered until a value has been returned. This mechanism implicitly adds synchronization between processes. Two primitives (*Wait* and *Awaited*) provide for explicit synchronization.

Syntax:

```
v = p -> f ( parameters );
  ...
v.foo; //Triggers a wait if awaited
if ( Awaited (v) ) { //Test the status
Wait (v); //Triggers a wait if awaited
```

Sharing

The semantics of communication between processes is a copy semantics for passive objects: all parameters are automatically transmitted by copy from one process to another. Of course, active objects are subject to a reference semantics: all processes are transmitted by reference from one process to another. As a consequence, shared objects (ie. passive objects which are shared between active objects) are not allowed as such. The implication of this is that any passive object which is to be shared between several active objects must be made an instance of a process heir. This is simply achieved by inheriting from the process class, which also provides an automatic FIFO synchronization.

All the features of this library are programmed on top of the MOP we defined for C++, without any compiler modification.

5 Interconnection Networks

The interconnection network layer is in charge of the actual transmission of messages (objects in fact, due to the reification mechanism) from one process to another, or one machine to another machine on the network.

We are currently merely using the PVM system. However, the intended goal is to define more sophisticated communication policies in order to take advantage of hardware capabilities. For instance, we are working on the definition and implementation of communication algorithms adapted to various network topologies and hardware, automatic or semi-automatic mapping and load balancing strategies, and overlapping of communication-computations.

The MOP mechanism makes it possible to incorporate such optimisations within the libraries in a transparent manner for the final user.

6 Conclusion

The *SLOOP* project offers at the top a generic environment for distributed discrete event simulation. However, the two other parts of this three-layer system are also available as stand alone tools: the parallel object-oriented language C++//, and, in the future, a library of communication routines and mapping.

References

[1] I. Attali, D. Caromel, and M. Oudshoorn. A formal definition of the dynamic semantics of the Eiffel language. In *Sixteenth Australian Computer Science Conference (ACSC-16)*, pages 109–120, February 1993.

[2] I. Attali, D. Caromel, and A. Wendelborn. A formal semantics and an interactive environments for sisal. In Amr Zaky, editor, *Tools and Environments for Parallel and Distributed Systems*. Kluwer Academy Publishers, 1996. pp 231-258.

[3] F. Baude, N. Furmento, and D. Lafaye de Micheaux. Managing true parallelism in ADA through PVM. In *First European PVM Users' Group Meeting, Rome*, 1994.

[4] J.-C. Bermond and P. Fraigniaud. Broadcasting and gossiping in de Bruijn networks. *SIAM Journal on Computing*, 23(1):212–225, 1994.

[5] J-C. Bermond, P. Fraigniaud, and J. Peters. Antepenultimate broadcasting. *Networks*, 26(3):125–137, 1995.

[6] J-C. Bermond, L. Gargano, S. Perennes, A. Rescigno, and U. Vaccaro. Efficient collective communications in optical networks. Technical Report 95-65, I3S, 1996. to appear ICALP96.

[7] J-C. Bermond, L. Gargano, A. Rescigno, and U. Vaccaro. Fast gossiping by short messages. In *Proc.22nd ICALP95, Szeged, Hungary*, volume 944, pages 135–146. Lecture Notes in Computer Science, Springer Verlag, 1995.

[8] J.-C. Bermond, P. Hell, A. L. Liestman, and J. G. Peters. Broadcasting in bounded degree graphs. *SIAM Journal on Discrete Mathematics*, 5(1):10–24, 1992.

[9] J-C. Bermond and S. Perennes. Efficient broadcasting protocols on de bruijn and similar networks. In *Proc. Conference SIROCCO95 Olympie June 95*. Carleton U. Press, 1995.

[10] J.C. Bermond, P. Michallon, and T. Trystram. Broadcasting in wraparound meshes with parallel monodirectional links. *Parallel Computing*, 18:639–648, 1992.

[11] D. Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, November 1989.

[12] D. Caromel. Concurrency and reusability: From sequential to parallel. *Journal of Object-Oriented Programming*, 3(3):34–42, September 1990.

[13] D. Caromel. Programming abstractions for concurrent programming. In *Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90)*, pages 245–253, November 1990.

[14] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[15] D. Caromel. Abstract control types for concurrency, position statement for the panel : *How could object–oriented concepts and parallelism cohabit ?*. In *IEEE ICCL'94, International Conference on Computer Languages*, pages 205–214, August 1994.

[16] D. Caromel, F. Belloncle, and Y. Roudier. The c++// system. In G. Wilson and P. Lu, editors, *Parallel Programing Using C++*. MIT Press, 1996.

[17] C.Gavoille and S.perennes. Memory requirement for routing in distributed networks. Technical Report 95-37, I3S, 1995. to appear PODC96 (best student awrd paper).

[18] P. Ferrante, P. Mussi, G. Siegel, and L. Mallet. Object oriented simulation: Highlights on the PROSIT parallel discrete event simulator. In *Western Simulation Conference 1994, Tempe (AZ)*. SCS, January 1994.

[19] B. Gaujal, A.G. Greenberg, and D.M. Nicol. A sweep algorithm for massively parallel simulation of circuit-switched networks. *Journal of Parallel and Distributed Computing*, 1993.

[20] G. Kiczales. Why are black boxes so hard to reuse? Towards a new model of abstraction in the engineering of software. In *OOPSLA'94, Invited talk*, 1994.

[21] 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.

[22] Philippe Mussi and Hery Rakotoarisoa. Parseval : A workbench for queueing networks parallel simulation. In *Modelling and Simulation FSM 93*. Society for Computer Simulation, 1993.

[23] J. G. Peters and M. Syska. Circuit-switched broadcasting in torus networks. Technical Report CMPT TR 93-04, Simon Fraser University, 1993. to appear in *IEEE Transactions on Parallel and Distributed Systems*.

[24] J. De Rumeur. *Réseaux d'interconnection*. Masson, dec 1994. to be translated.

[25] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.