Evolutive Prototyping of Heterogeneous Distributed Systems Using Hierarchical Algebraic Petri Nets

D. Buchs and J. Hulaas
Department of Computer Science
Swiss Federal Institute of Technology
CH-1015 Lausanne

ABSTRACT

This paper presents several extensions to a new evolutive prototyping process, called *object-oriented mixed prototyping*. Our application field is complex distributed systems modelled by the means of CO-OPN, a hierarchical algebraic Petri net language. CO-OPN being a strong synchronous language, we also show an original implementation scheme for its high-level communication model in heterogeneous wide-area networks.

1. INTRODUCTION

The advantage of formal methods resides by definition in their unambiguous nature. This allows them to be supported by tools which automatically verify the properties, extract useful information or transform them into other representations. The benefits of formal specifications become even more evident when they can be used continuously through several stages of the software development cycle, like specification, implementation, integration or testing.

Our purpose is to fill the gap that currently exists between the phase where a specification is simulated, generally using some logical inference tool, and the phase where the modelled system has an *efficient and maintainable* implementation. This objective is realized by application of a methodology we call *object-oriented mixed prototyping* (OOMP), which is based on the facilities of object-oriented programming languages such as polymorphism and dynamic binding.

Our work is done within the frame of the $\text{CO-OPN}_{1.5}$ (Concurrent Object-Oriented Petri Nets) specification language [6][7] and the associated development tool called SANDS (Structured Algebraic Nets Development System) [8][9]. CO-OPN is directed towards specification of complex distributed systems. The specification language is based on hierarchical algebraic Petri nets [21], where the Petri nets are used to model operational and concurrent aspects. The object notion of CO-OPN serves as a structuring means, and naturally captures the concurrent aspects of a model. Each CO-OPN object is a Petri net, which is connected to other objects through transaction-oriented synchronization links.

In this paper, we consider the way we can build incremental prototypes for concurrent systems specified with CO-OPN by using an object-oriented programming language for the implementation. We show that the use of object-oriented programming languages, compared to previous approaches [11], allows for more flexibility in the incremental prototyping process, and gives better guarantees that the final prototype conforms to the formal specification.

We made this experience with the object-oriented part of the Ada95 language [1] but the principles can be transposed to any modern object-oriented programming language. The Ada95 language was chosen for its commitment to strict software engineering principles: Strong and static typing, generic modules and information hiding are the securities our prototyping methodology relies upon. These possibilities allow a very straight-forward mapping of the specifications into the implementation level. Another advantage of Ada95 in the context of formal specifications is that its semantics have been normalized, which prevents the use of ambiguous constructs.

In a first part we explain the advantage of using prototypes and the object-oriented techniques for building such prototypes. Then we show in which sense concurrency aspects can be incorporated into such a framework. Finally we give some indications about the algorithmics involved with the implementation of CO-OPN specifications.

2. THE PRINCIPLES OF MIXED PROTOTYPING

Prototyping can be applied with different objectives and at different levels of the development process, for instance for determining the feasibility of a project, for establishing and validating the requirements or for validating the functional specifications.

According to the taxonomy of [2], *evolutive prototyping* is a process where the prototype evolves to the real end-user application instead of being merely thrown away after use. This implies that the prototyping tool provides for acceptable

---

This work was funded by the Swiss Federal Institute of Technology of Lausanne, the European Esprit Long Term Research Project 20072 “Design for Validation” *(DeVa)* through the Swiss OFES (Office Fédéral pour l’Education et la Science), and the Swiss National Science Foundation project 2000.40583.94 “Formal Methods for Concurrency”.

characteristics concerning flexibility, completeness, speed and robustness. The prototypes developed from CO-OPN specifications follow this strategy, since they constitute *stand-alone* Petri net simulators.

In the field of evolutive prototyping, *mixed prototyping* is a methodology for incrementally replacing abstract modules, i.e. automatically generated from formal specifications, with concrete ones, i.e. hand-coded. This can be done while continuously preserving the possibility of testing the modified prototype: Figure 1 illustrates a hierarchy of modules under development where some modules are abstract, some completely concrete, and some others in an intermediate state, called semi-abstract. This situation still allows to execute the prototype, provided that it is in a minimally coherent state. The concept of mixed prototyping was first introduced by Choppy [10].

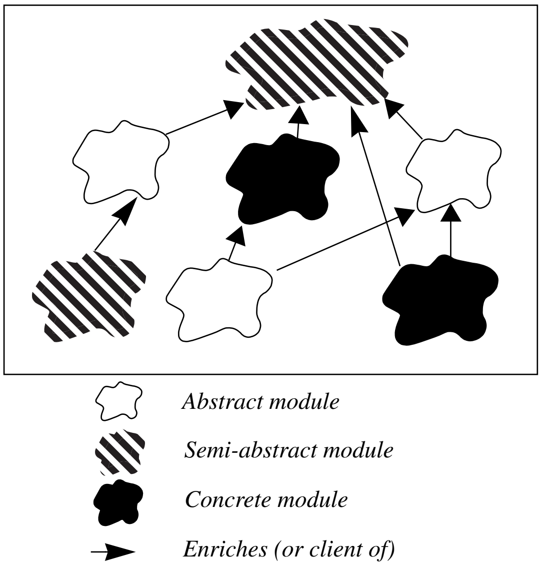

Figure 1: The concept of mixed prototyping

The motivation behind mixed prototyping is that a specification compiler can never be efficient or general enough to cover all the cases encountered in real-world systems. Moreover, the developer may want to add functionalities, e.g. customized I/O, that cannot be specified using a particular formalism. The prototyping tool, when producing the initial implementation of the specifications, may generate a highly optimized and thus hardly legible code. The abstract modules are however not meant to be read or modified by humans. Rather, the intent is to give the opportunity to add pieces of software that are automatically and harmoniously incorporated into the environment.

3. THE OOMP METHODOLOGY

Until now, mixed prototyping had only been practised in the context of algebraic specifications by resorting to Ada83 for its support for genericity. Our contribution is that we exploit the flexibility offered by object-oriented implementation languages and that we apply these results to the frame of distributed systems described by Petri nets. From now on we will talk of abstract classes, symbolic classes, and concrete classes (Figure 2 depicts this organization). The abstract and symbolic classes are generated automatically, the former containing all the algorithmic information, and the latter providing an internal representation, the structure of which is generic.

Use of object-orientation adapts advantageously the principles of mixed prototyping to modern programming languages and paradigms. To this end, we exploit the notion of multiple implementations for a single abstraction, as well as a mixture of class-based and prototype-based coding style. The general programming techniques employed to enable this prototyping process is described in detail in [16]; here, in contrast, we explain the algorithmics involved in the CO-OPN specific implementation.

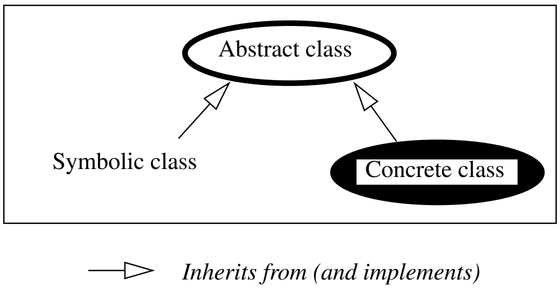

Figure 2: Object-oriented view of mixed prototyping

Compared to previous results, we can state the following additional improvements:

- The programmer autonomously redefines and specializes operations of the abstract class. The prototyping tool does not need to know what changes have been done, only which class to use as implementation.
- There is no hidden mechanism: We exploit the natural resources of the object-oriented paradigm. The prototyping process gains transparency.
- Client classes need not be modified nor even re-compiled when the developer switches forth - and even possibly back - between the symbolic and the concrete classes: The methodology reduces the impact of change through the use of prototype objects.
- Memory management is transparent to the client modules. The abstract class can even totally discharge its derived classes of this burden.
- The problem of communicating algebraic abstract data types in heterogeneous distributed systems, which we address by considering signatures as an interface description language (IDL), may also be optimized by redefining the

corresponding operations in concrete classes. More information on this subject may be found in [15].

- The prototyping tool can automatically generate sub-classes of the concrete class, which compare, under certain restrictions, the results of the abstract operations - which are supposed correct - with those of their concrete versions, and thus provide a form of dynamic validation.

Figure 3 illustrates the OOMP process, and enhances the fact that the developer proceeds incrementally, starting with the generation of abstract code and ending when the internal representation and all the operations have been replaced by hand-written code.

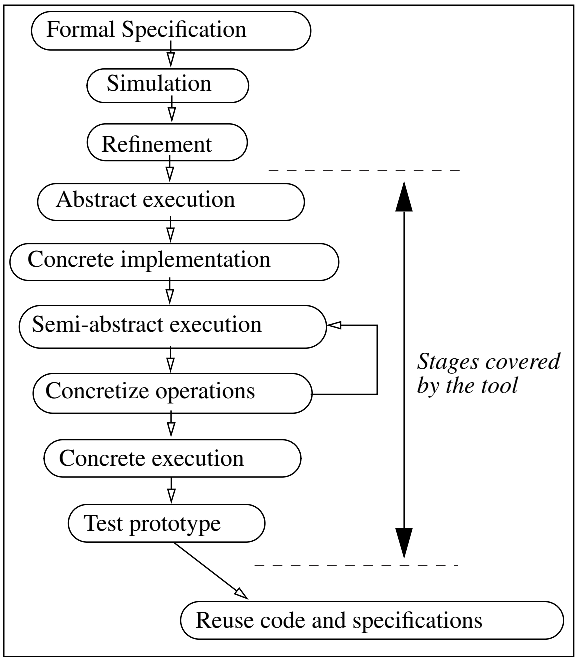

Figure 3: The mixed prototyping process

4. THE IMPLEMENTATION STRUCTURE

The OOMP methodology mainly defines how to interface correctly the different components of the system. On the other hand, it does not impose any particular operational semantics on the prototype. Therefore, the prototyping tool just has to settle a structure which is compatible with the development process, i.e. which allows the user to progressively replace the internal representation and to redefine the operations inherited from the abstract class. This is realized with the support of polymorphism. Dynamic binding is employed as a means of selecting a new implementation according to a given configuration without having to modify or recompile client modules during the prototyping process.

When considering specification languages including concurrent objects such as CO-OPN, it is necessary to include other concerns about the development of mixed prototypes. Following the well admited idea that concurrency control and object functionnalities must be managed differently within concurrent object languages, we split each concurrent object into two classes called *Control* and *Action* (see Figure 4).

Role of the control class

The control class is designed to ensure a correct implementation of the distributed simulation. It literally serves as interface between the clients and its own action class, which provides the actual functionality of the CO-OPN object. This organization allows the evaluation of the synchronization conditions to be decomposed into elementar operations offered by the action class: *Precondition*, *Transition* and *Postcondition*.

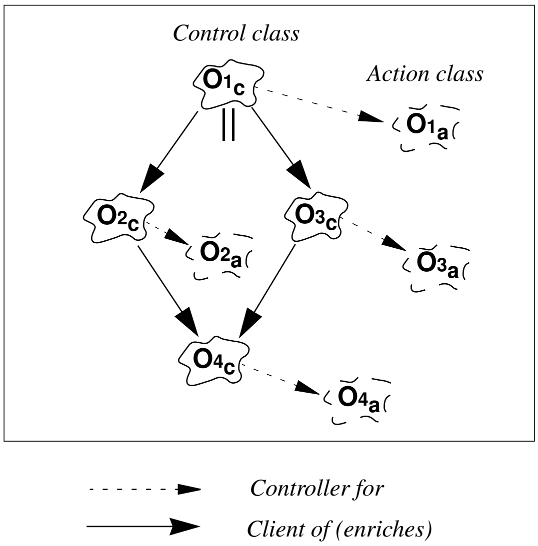

Figure 4: Hierarchy in mixed prototyping for concurrency

The control class has its interface defined by the specification but can be implemented through different algorithms depending on the kind of support it must ensure: This may depend on the topology of the specification hierarchy, as well as the complexity of the synchronization axioms. Different levels of parallelism may be selected: The CO-OPN semantics specify that propagation and evaluation of synchronization conditions are viewed as blocking method calls, but optimizing implementations should of course try to perform as much as possible in parallel, while keeping the same observable behaviour.

Coarse grain parallelism is needed for the execution of each CO-OPN object, an object being a sub-net of arbitrary size defined by the programmer. These units evolve naturally in a loosely coupled fashion. They are very similar to *process-based Prolog* modules [5], in that their interface is composed of predicates - the CO-OPN methods - and that they must provide support for distributed backtracking. The main distinctive property of control classes versus the few distributed Prolog systems which offer distributed backtracking[1], resides in the treatment of shared objects, which is described in the following section.

The Implementation of Action Classes
The prototyping tool compiles the specifications for an internal abstract machine which then has to be mapped onto a concrete machine, ranging from heterogeneous networks of workstations to massively parallel supercomputers. The different phases of the compilation are:

- Identify the direction of the data flow in the axioms and require all implicit conditions to be made explicit. This will often have to be completed with externally supplied annotations since the CO-OPN language does not provide a syntax for indicating the mode of method parameters.
- Generate code for pattern-matching on algebraic abstract data types by following Schnoebelen's compilation algorithm [23].
- Merge into single functions the preconditions, transitions and postconditions gathered from the different axioms defining each method. The compilation should mask as much as possible the notion of axiom: The larger the operations of the abstract code, the better the potential for optimization in the concrete code.
- Decompose synchronizations into sequences of calls to the corresponding preconditions, transitions and postconditions. The names of the resulting operations are respectively `pre`, `sync`, and `post`.

5. OPERATIONAL SEMANTICS

In this section we give a short description of the run-time support and modular concurrency control provided by the control class. We will concentrate on inter-object communication and concurrency control, as the intra-object behaviour, which is described by Petri nets, is less innovative in its implementation (the data structures are explained in [16]).

Synchrony in wide-area networks
CO-OPN is a strong synchronous language [6], like Esterel [3] and Statecharts [17]. This means that every operation, or change of state, be it internal or external, is viewed as being instantaneous[1]. In the case of languages like Esterel, this property was designed as a means of mastering and reasoning about reactive real-time systems. In CO-OPN, internal transitions being instantaneous is due to the particular interpretation of the Petri nets as a synchronous Petri nets model. But, concerning the methods offered by CO-OPN objects, the objective is to have a convenient way of abstracting away the behaviour of external components. This modelling facility is also supported by a solid formal basis [14].

In distributed systems, however, synchrony is difficult to simulate, because of the lack of absolute time reference and the non-negligible communication medium. Some work has been done to implement Esterel on local-area networks by maximizing the communication and reaction time for every event, see for instance [4]. The principles we elaborate here for CO-OPN are valid for wide-area networks like Internet, since we do not make any assumption about the performance of the system: We only try to interpret the effects of synchrony on the semantics of method calls in CO-OPN. From the requirement of synchrony it follows that the synchronization primitives of CO-OPN receive the following particular semantics:

- The *sequence operator,* noted `M1..M2`, requests that method `M1` be evaluated before `M2`, and that the global system state at the end of `M1` be taken as input state for `M2`. This leaves no room for interleaving third-party transactions in-between in order to increase concurrency, unless it can be proven that they will only affect disjoint parts of the system. More operationally, the sequence is translated into the consecutive operations `M1.pre`, `M1.sync`, `M1.post`, `M2.pre`, `M2.sync`, `M2.post`.
- The *simultaneity operator*, noted `M1 || M2`, requires methods `M1` and `M2` to be evaluated strictly at the same time, which is a stronger condition than the usual notion of concurrency. This means that whenever an object is shared between `M1` and `M2`, its internal ressources will also have to be shared between both partners. The shared object will appear as being split during the whole existence of `M1` and `M2`, before being merged again at their end. From a low-level point of view, simultaneity will be scheduled as follows: `M1.pre, M2.pre, M1.sync, M2.sync, M1.post, M2.post`. Each pair `M1.op`, `M2.op` may be interchanged since the `||` operator is commutative.

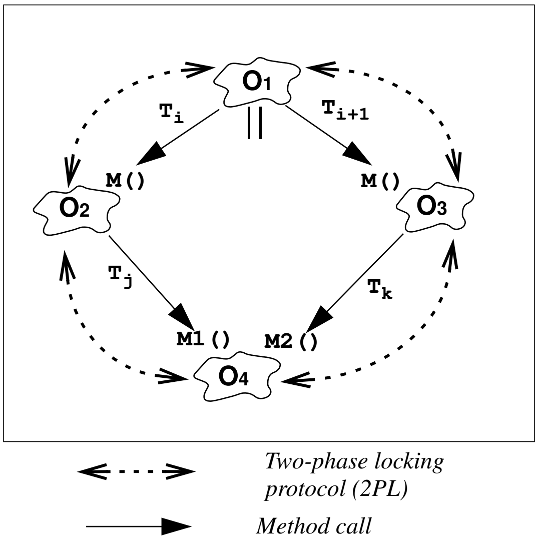

Figure 5: Simultaneity with shared object

A transaction-oriented system
Another salient property of CO-OPN models is that they always do their best to satisfy a request. To this end, an

1. To our knowledge, only Delta-Prolog [20] and CS-Prolog [13] offer built-in distributed backtracking.
1. Here *synchronous* is not taken in the sense of blocking method calls.

exhaustive enumeration of all token combinations of input places and all firable transitions and methods is performed until a solution is found which all the partners of the synchronization will agree upon. If this search fails, the previous system state will be restored as if nothing had happened. This latter all-or-nothing behaviour is called *atomic*, or *transaction-oriented*, in object-based distributed systems. Our implementation is in fact an extension of the notion of nested transactions with two-phase locking protocol (*2PL*) to ensure atomicity and consistency [19]. This protocol holds for CO-OPN in the general case where method calls involve no object sharing between simultaneous requests.

As described on Figure 5, the shared object `O4` must be informed that the incoming invocations are to be processed simultaneously: This information may not be compiled in statically, since that would break the modularity and potential for code reuse. Object `O4` must therefore receive enough information to reestablish locally a complete view of the hierarchy of nested method calls above itself. The data directly contained in the locks generated by the 2PL protocol gives the list of parent nodes and transactions involved. To this we just need to indicate additionally the nature of the synchronizations at each level of the invocation graph. This gives the following for object `O4`:

- Incoming invocation `O4.M1()` provides 2PL lock list [$O1.T_i$, $O2.T_j$] and synchronizations [`Sim`, `Basic`]
- Incoming invocation `O4.M2()` provides 2PL lock list [$O1.T_{i+1}$, $O3.T_k$] and synchronizations [`Sim`, `Basic`]

where $O_i.T_j$ is the name of transaction T_j created by object O_i for calling a method, and where `Sim`, `Seq` and `Basic` indicates the kind of synchronization for the corresponding sub-transaction in the 2PL lock list.

Communication delays and processing needs will systematically infer variable arrival times at shared objects for the different partners of a simultaneous request. We have therefore adopted an optimistic attitude towards simultaneity by assuming that objects are not shared until the system evolution proves the contrary. In Figure 5, say the invocation `M1` from `O2` arrives first. Object `O4` does not know whether other requests will have to be processed simultaneously[1], and therefore chooses to service immediately the ones it already has knowledge of. When the invocation `M2` from `O3` comes in, object `O4` will find out that it should have been processed simultaneously with the one from `O2`. By looking backwards at the state it had when the simultaneity was supposed to take place, the shared object will decide if the synchronization could have happened, or if it might have succeeded by changing a posteriori the result of `M1`. To this end, we have extended the 2PL protocol to allow committed sub-transactions to either re-commit with another result, or to fail. The messages which announce new results for such already committed transactions are called *antimessages* (Figure 6).

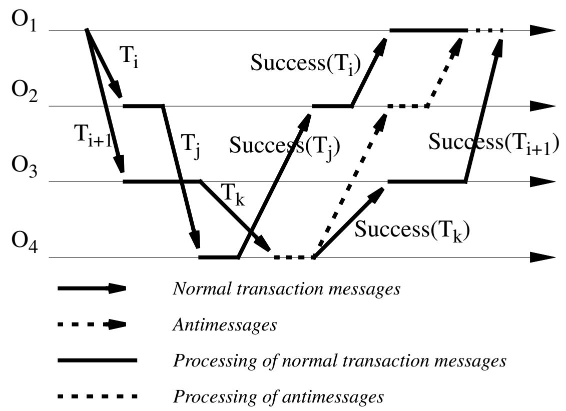

Figure 6: Optimistic simultaneity with antimessages

This of course implies that the state of the system is saved in a log at the start of each sub-transaction, so that it can be restored at any moment. We must also record all the side-effects, such as sending messages, in order to abort them during the backtrack. The log will keep growing as successful invocations add up, and can only be purged when the top-level transaction ends, i.e. when global commitment, with a protocol such as the two-phase commitment (*2PC*) [19], is over.

Interfacing of CO-OPN prototypes

If we want to enable the use of CO-OPN-specified prototypes in a more general context where external components are not formally specified, we will need an interfacing mechanism. The role of this interface is to package the CO-OPN transactions to make the whole prototype appear as a single object to the outside world. The format of CO-OPN transactions is not totally compatible with classical nested transactions, but we can produce an abstraction where the prototype is seen as a simple node in the nesting structure.

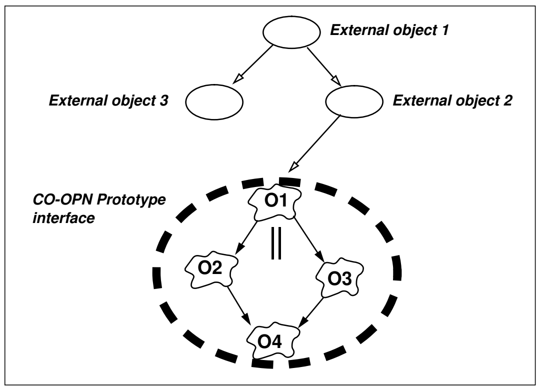

Figure 7: A CO-OPN prototype within an existing system

In Figure 7, the system described in Figure 6 is inserted into an existing configuration by providing a surrounding interface.

1. Notice that not even the emitter `O1` can know whether its request will involve indirect object sharing.

This principle is currently investigated by other researchers in the field of federated databases [22].

6. PROPERTIES OF RESULTING PROTOTYPES

We have shortly sketched the principles that are applied to incrementally build prototypes from abstract specifications. It seems that the results concern principally the domain of software engineering for concurrent systems, in particular the improvement of the software development process. By adopting our methodology, users can considerably improve the global quality of the system under development. Nevertheless, several questions remain open: We can mention the evaluation of the real performance that can be expected from a prototype compared to a completely hand-written application. Some efficiency may have to be sacrificed by following the design patterns systemized with object-oriented mixed prototyping. This flaw will however slowly disappear as development tools for object-oriented languages get more performant. Let's not forget that some of these utilities already exist, such as the `flat` tool, which collapses Eiffel inheritance hierarchies at the source level.

7. CONCLUSION

In this paper we have shown how object-orientation gives a new dimension to mixed prototyping, and in particular that concurrent aspects described by hierarchical Petri nets can be included in this framework: It is more intuitive because it uses the natural resources of object-oriented programming. It brings more flexibility, allows a shorter development time, and gives better guarantees for the correctness of the concrete code. We also presented the run-time support for the distributed execution of CO-OPN, taken as a very high-level programming language. We hope by our work to contribute to a more widespread application of hierarchical Petri nets and formal methods in general.

8. REFERENCES

[1] *Ada Reference Manual, Language and Standard Libraries,* ISO/IEC 8652:1995.

[2] S. Asur and S. Hufnagel, "Taxonomy of Rapid-Prototyping Methods and Tools", *4th International Workshop on Rapid System Prototyping,* Research Triangle Park Institute, North Carolina, USA, June 1993.

[3] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". Technical Report 842, INRIA, May 1988.

[4] F. Boniol and M. Adelantado, "Programming Distributed Reactive Systems: a Strong and Weak Synchronous Coupling", *7th International Workshop on Distributed Algorithms WDAG'93,* LNCS 725, Sept. 1993, pp 294-308.

[5] K. De Bosschere, "Process-based Parallel Logic Programming: A Survey of the Basic Issues", *11th ICLP: Post-Conference Workshop W1 on Process-Based Parallel Logic Programming*, Sta. Margherita, Italy, June 1994.

[6] D. Buchs, N. Guelfi. CO-OPN, "A Concurrent Object Oriented Petri Net Approach". *12th Int. Conf. on Application and Theory of Petri Nets*, 1991, pp. 432-454.

[7] D. Buchs, O. Biberstein, M. Buffo, C. Buffard, J. Flumet, J. Hulaas, G. Di Marzo, P. Racloz, "SANDS1.5/CO-OPN1.5: An Overview of the Language and its Supporting Tools", Technical Report #95/133, DI-EPFL, 1995.

[8] D. Buchs, J. Flumet, P. Racloz, "SANDS Structured Algebraic Net Development System", Research report no 71, CUI 1993, also in 14th Int. Conf. Th. Petri Nets, Tool Presentation, Chicago, 1993, also in 16th International Conference on Application and Theory of Petri Nets, Torino, 1995, pp 45-53.

[9] D. Buchs, P. Racloz, M. Buffo, J. Flumet and E. Urland, "Deriving Parallel Programs Using SANDS Tools", Transputer Communication Journal, vol. 3, no. 1, 1996, pp. 23-32.

[10] C. Choppy, "Formal Specifications, Prototyping and Integration Tests", Proc. of the 1st European Software Engineering Conference, 1987.

[11] C. Choppy, S.Kaplan, "Mixing Abstract and Concrete Modules: Specification, Development and Prototyping", Proc. 12th Int. Conf. on Software Engineering, Nice, 1990.

[12] H. Ehrig and B. Mahr, "Fundamentals of algebraic specification 1: equations and initial semantics", EATCS Monographs, Springer Verlag, 1985.

[13] Sz. Ferrenczi and I. Futo, "CS-Prolog: a Communicating Sequential Prolog", in P. Kacsuk and M. Wise editors, *Implementations of Distributed Prolog,* pp 357-378, John Wiley & Sons, Chichester, 1992.

[14] N. Guelfi, "Les Réseaux Algébriques Hiérarchiques: un formalisme de spécifications structurées pour le développement de systèmes concurrents", PhD thesis, Université d'Orsay Paris XI, 1994.

[15] J. Hulaas, "Introduction to Object-Oriented Mixed Prototyping with Algebraic Specifications", Technical Report 95/155, EPFL-DI-LGL, 1995.

[16] J. Hulaas, "An Evolutive Distributed Petri Nets Simulator", 10th European Simulation Multiconference ESM'96, Budapest, Hungary, 2-6 june 1996, pp. 348-352.

[17] D. Harel, "Statecharts: a Visual Formalism for Complex Systems", Science of Computer Programming, 8(3):231-275, 1987.

[18] B. Meyer, "Eiffel: A Language and Environment for Software Engineering", Interactive Software Engineering, Inc, 1987

[19] J.E.B. Moss, "Nested Actions: an Approach to Reliable Distributed Computing", PhD thesis, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[20] L.M. Pereira, L. Monteiro, J. Cunha, and J.N. Aparício, "Delta-Prolog: A Distributed Backtracking Extension with Events", in G. Goos and J. Hartmanis, editors, *Proceedings of the Third International Conference on Logic Programming*, volume 255 Lecture Notes in Computer Science, Springer Verlag, London, July 1986.

[21] W. Reisig, "Petri Nets and Algebraic Specifications", Theoretical Computer Science 80, pp. 1-34, 1991.

[22] W. Schaad, H.-J. Scheck, G. Weikum, "Implementation and Performance of Multi-level Transaction Management in a Multidatabase Environment", *5th Int. Workshop on Research Issues in Data Engineering RIDE-DOM,* Taiwan, March 1995.

[23] Ph. Schnoebelen, "Refined Compilation of Pattern-Matching for Functional Languages", Science of Computer Programming, 11:133-159, 1988.

[24] P. Wegner, "Dimensions of Object-Based Language Design", in OOPSLA '87 Conference Proceedings, pp. 168-182, 1987.