Computer-Aided Formal Specification for Concurrent Engineering Platforms

R. Guetari and G. T. Nguyen
INRIA Rhône-Alpes
655 Avenue de l'Europe
38330 Montbonnot — Saint Martin — France

Abstract
Formal methods, techniques and tools are, at the present time, an active research topic in different areas of computer science (knowledge representation, real-time systems, algorithms, etc.). These formal techniques are intended to help users specify consistently their needs and verify them. Only mathematical techniques are able to prove or to verify the coherence of the specification of a given system or algorithm, etc. However, there is an enormous difficulty to put into use the mathematical techniques and concepts. This difficulty stems from the fact that these mathematical techniques and concepts are accessible only by a minority of specialists. To solve this problem, we have to develop tools and methods to help users to make the most out of formal approaches, without the apparent complexity of mathematical problems.
This paper presents a tool, called CAST (Computer-Aided Specification Tool) dedicated to help users specify communicating processes and systems in concurrent engineering environments. CAST is a graphical tool which provides a friendly user interface. At the present time, CAST allows users to specify design processes by representing them in the form of automata and provides an SCCS (Synchronous Calculus of Communicating Systems) specification. This tool is developed in the SHOOD project which aims at providing tools and methods for the integration of engineering design systems.

Keywords
Concurrent engineering, formal specification, Object-oriented design, Cooperative engineering systems

1. INTRODUCTION

The design of integrated concurrent engineering platforms has received much attention in recent years, because competition strives for shorter design delays and manufacturing costs among competing firms (Dertouzos 1989). Concurrent engineering allows to design products by taking into account downstream concerns, such as manufacturability, testability and maintainability of the designed products. As artifacts become increasingly sophisticated and as competition becomes more global, it has been recognised that design should be a cooperative endeavor carried out concurrently by many agents with diverse kinds of expertise, thus lead to shorter design to market delays. It requires, however, advanced coordination and integration capabilities (Brown 1992). One way to assist the design engineers in such environments is to provide flexible collaborative frameworks. They allow various teams to work simultaneously and consistently on different parts of a global project (Cutkosky 1993 ; Tenenbaum 1992). But the existing software, tools and computer platforms used in manufacturing enterprises require powerful, versatile and open architectures to take into account these legacy systems (McGuire 1994 ; Paul 1995). A potential technological breakthrough consists in developing generic

integration platforms that provide high-level distributed services (Genesereth 1992b), i.e., at the applications' knowledge level (Newell 1982).This allows the various tools to communicate and cooperate through high-bandwidth networks of distributed computers (Genesereth 1992a). These machines offer specific sharable services. One of the first approaches in this area was the ARPA's Knowledge Sharing Initiative (Patil 1992).
A requisite for developing such platforms is the specification of a model for a generic design process, allowing the consistent cooperation of the distributed services.
SHOOD (SHared Object-Oriented Design) (Nguyen 1993) explores an approach based on the monitoring of the evolution of design objects, following the engineers' design decisions. The idea is to track closely the design path followed by the engineers, and to provide a suitable reactive model of the design artefacts. The reactive object model is designed to evaluate the consequences of the design actions. The latter are modeled as modification requests, that are sent simultaneously by the various teams working on a concurrent engineering platform. This approach capitalizes on specification and verification techniques developed for reactive systems (Jourdan 1994).

2. OVERVIEW OF SHOOD

The goal of the project SHOOD is to provide tools and methods for the integration of engineering design systems. SHOOD is at the same time a knowledge representation system and a platform which provides tools and methods for integration of engineering design systems. SHOOD proposes three work-packages allowing: (i) the definition of a generic product model (based on knowledge representation techniques), (ii) the formal specification of a reactive design model (Nguyen 1996) and (iii) the development of an integration platform supporting cooperative engineering (Guetari 1996).

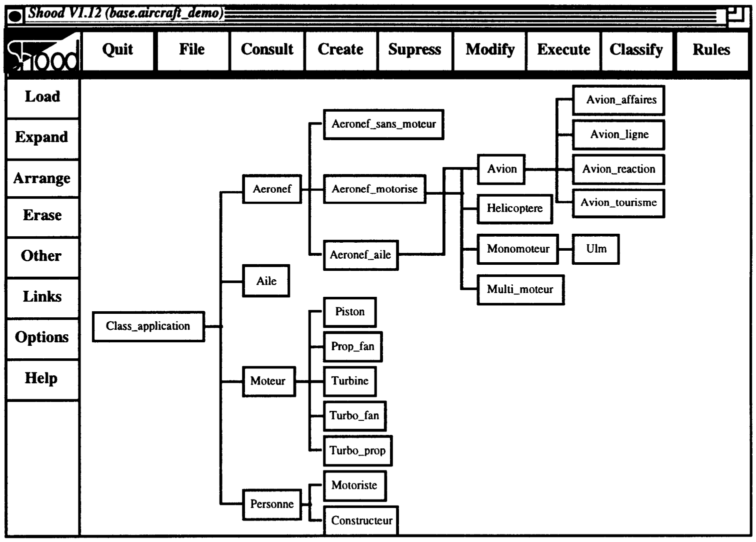

Figure 1 - SHOOD's user interface

As a knowledge representation system, SHOOD is based on object-oriented paradigm (Nguyen 92a) and supports dynamic, partially inconsistent and possibly incomplete knowledge. The model uses a reflexive object model (Bounaas 1995 ; Nguyen 1991) and implemented with a metaclass kernel that defines all the basic concepts. Over and above the metaclass and class/instance concepts, SHOOD allows the multiple inheritance, specialisation, composition and

dependency relationships (e.g., existential and sharing dependencies, object versions' dependencies), generic functions using method specialisation, active rules making SHOOD a powerful active-objects store and a classification mechanism for composite objects. The knowledge representation system provides an X-11 based easy and friendly user interface (figure. 1) allowing the access to the object definitions and manipulations.
The SHOOD's platform of integration proposes: the formal specification of a reactive design model (Nguyen 1995) and the development of an integration platform, that supports concurrent engineering of the design products.
The reactive object model implemented by SHOOD supports three basic requirements of engineering design applications (Nguyen 1991):
• the evolution of design components during the design of products,
• the multiple representations of the objects being designed concurrently by various teams (Nguyen 1992b),
• the complexity of the product structures, often considered as intricate "part-of" relationships, but considered also here as tightly inter-dependent components. It entails that semantic relationships, e.g., existential dependency relationships among components, are considered.
Modelling the design process requires extensive knowledge on the activities involved, and a formal background on which to elaborate a suitable model (Gero 1993). Yet, design decisions can be erratic and amenable to multiple trial and error cycles. Also, the intrinsic dynamics of the design process makes it difficult to comprehend and control them using formal techniques.
The process model developed for SHOOD is intended to provide a powerful and flexible framework for capitalising on design experience and achieve new engineering projects. It supports incremental construction of process models and their dynamic modification to adapt to the projects' specific goals. It is based on partially ordered sets of tasks and scheduling operations.

3. OVERVIEW OF SCCS

Process models are specified using the SCCS algebra (Boudol 1985 ; Milner 1989). SCCS is based on a double structure:
• a set **Act** of atomic actions (i.e. temporally indivisible and lasting “one instant"). Milner defines the simultaneous composition of actions “.”.
• a set $\mathbf{P}$ of agents, which can be seen as the behaviours of processes.
The basic concept of the calculus is represented by the expression: $P \xrightarrow{a} P'$, which means that the agent P is able to transform into an agent P' by performing an action a. Let us focus on the “action” and “instant” concepts. The instant of an electronic transition process is perhaps a nanosecond, whereas it is a millennium for the geophysician. Indeed the instant of a system is the duration of the elementary action (or event) of this system. Instant and clock are basic concepts of every synchronous model of time. Milner defines the following “basic” operators on the set of agents.
• The *action operator* prefixes the behaviour of an agent E with an action a. Its expression is **a:E**. This expression specifies the agent which executes the action a before carrying the behaviour of E.
• *Sum*: (E_i) is an agent family. $\sum_i E_i$ (or $E_1 + E_2 + ...$) defines the non determinist choice between the E_i’s behaviours. $E_1 + E_2$ is the agent where behaviour is E_1 behaviour or E_2 behaviour. An expression without the sum operator is called determinist i.e. after each action performed, an agent has a unique behaviour to adopt. The behaviour described by this kind of expressions is completely foreseeable.
• *Product*: $E \times F$ is the synchronous parallel composition of E and F. When E performs a and F performs b, then $E \times F$ performs the action a.b.
• *Restriction*: $E\lceil A$ shows the agent obtained from the behaviour of E, inhibiting all actions not members of a given set A. This operator is used to specify the synchronisation between processes.
• *Recursive operator*: this allows the designation of a family of agents, mutually and recursively defined by a system of equations. For instance let us consider a two-legged robot, which can put forward its left foot (action “l”) and its right one (action “r”). The *walk* process can be modelled by the agent: W = l:r:l:r... + r:l:r:l..., but this is not correct. The operator “fix” allows us to write it correctly:

fix(X = l:r:X). Then the *walk* process is specified by the agent:
W = fix (X = l:r:X) + fix (X = r:l:X)

- *Morphism*: this renames actions thanks to an endomorphism Φ. The expression <Φ>E designates the agent whose behaviour is obtained from E by replacing every action a by Φ(a).

We also use derived operators which are defined using the basic operators. We have especially derived the delay "δ" which sets an undefined wait, the sequence operator ";"... The full description of SCCS and derived operators is out of the scope of this paper. It is detailed in (Milner 1989 ; Guetari 1995).

Example: Let P a design process represented by the tasks T1, T2, T3 and T4. The tasks T1 and T2 are executed simultaneously, i.e., in parallel. T3 is realised after T1 and T2. Finally T4 is executed at the end of T3. If the task T4 ends abnormally, we go back to the tasks T1 and T2. The SCCS expression specifying P is:

BEHAVIOUR (P) = **H & start** fix (X = δ start ⇒(T1 × T2); T3 ; T4)

H & start means that the process P waits for the event **start** to be executed. This event may be unified with an event generated by a system clock (**H**) or emitted by the process P itself. The unary"⇒" operator specifies a triggering action. If s is an action and P is an agent, whose first action is a, the expression s ⇒ P means that the action s is executed simultaneously with action a. Thus s triggers agent P.

The task T1 consists in two sub-tasks T11 and T12 which are basic tasks and in turn executed simultaneously. Basic tasks in an SCCS expression are specified in the following form:

δ preAC: δ postAC

- *preAC* represents the event triggering the task,
- *postAC* is the event emitted when the task ends,
- the first delay operator (δ)indicates that the task waits for an event (*preAC*) which starts it,
- the second delay operator represents the duration needed to execute the basic task.

If we suppose that the tasks T2, T3 and T4 are basic tasks, the SCCS expression representing the global design process is :

BEHAVIOUR (P) = H & start fix (X = δ start ⇒ (((preT11: δ postT11) ×
(preT12 : δ postT12 : 𝟙)) × (preT2 : δ postT2)) : postT : preT3 : δ postT3 : preT4 : δ postT4 :
(testOK : 𝟙 + testNOK⇒start))

testOK is the event emitted if the task T4 ends correctly, testNOK is emitted if T4 ends abnormally.

4. CAST

The tool CAST is developed to help designers make the most out of SCCS algebra without its mathematical and theoretical aspects. Cooperative engineering design processes are graphically modeled in the form of automata. The graphic formalism of CAST is close to the ARGONAUTE environment, dedicated to the ARGOS language (Maraninchi 1992 ; Jourdan 1994). This formalism is based on hierarchical and parallel composition of automata. CAST offers a friendly user interface which allows the designer to build automata representing SCCS agents, and to combine them using a graphical formalism corresponding to SCCS operators (Fig. 2).

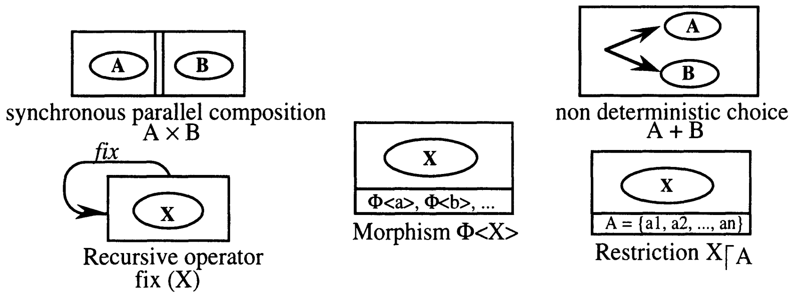

Figure 2. CAST graphic formalism

An SCCS agent is represented by an automaton where the transitions are the actions that the agent is able to perform. Each transition is parametrised by a set of inputs and a set of outputs, which are boolean values. The inputs enumerate the events which have to be present or missing in order to trigger a transition. The outputs represent the events generated when firing the transition.

4.1. Specifying Cooperative Processes

This section introduces the use of CAST through the example presented in the section 3. The tasks T1 and T2 are executed simultaneously, i.e., in parallel. T3 is realized after T1 and T2. Finally T4 is executed at the end of T3. If a problem is detected during the flight tests, we go back to the tasks T1 and T2. This process is modeled in CAST by the graph represented in figure 3.

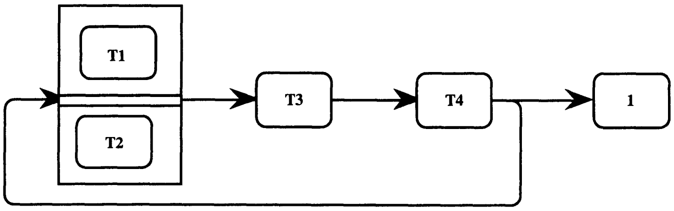

Figure 3. - A CAST-graph for the process P (section 3)

The different tasks in the design process may be complex. We have to divide each one into a partially ordered set of sub-tasks, until we obtain terminal tasks for basic design operations. For example, the task T1 of the process P consists in two sub-tasks: T11 and T12. T11 and T12 are in turn executed simultaneously ...

4.2. Editing Cooperative Processes

CAST allows us to hierarchically model cooperative engineering processes. The *File/New* command allows us to open a main window, in which we model the global automata representing the designed process. The different nodes of the automata represent the different tasks of the process. Figure 4 shows the CAST user interface. The automata represents the global model of an aircraft's design process.
If we select a given task (by a double click) a new window appears. In this window we must design the automata representing the selected task (Fig. 5 represents the task T1). Each task must be divided in turn, until obtaining basic tasks.
The *Edit* command allow us to modify an automataon (copy an automaton, cut a task, ...). Finally the *Compile* command provides an SCCS specification of the designed process. We can obtain an SCCS specification for each agent or an SCCS specification for the global process.

4.3. Verifying Cooperative Processes

It is necessary to specify correctly the process models in order to verify them. The verification consists in making sure that the process will not behave abnormally. A simple way to verify a process is to compare its effective specification with its ideal specification. This supposes that the ideal specification of a given process is known. A second way is to use a temporal logic to verify the specification of a given process. This method needs two different formalisms: the specification formalism and the verification one. The verification formalism must ensure that all the information provided by the specification will be considered. Another way to verify the specification of a process is given by observers. This method consists in specifying, for each process *P*, an observer process *OP* which behaviour consists in detecting illegal behaviours of *P*. The observer are also written in SCCS.

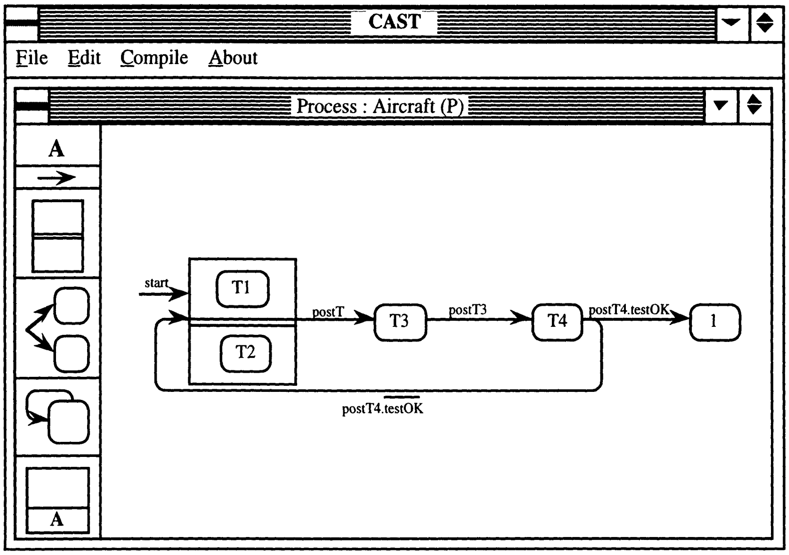

Fig. 4 - CAST user interface (global automaton for the process P)

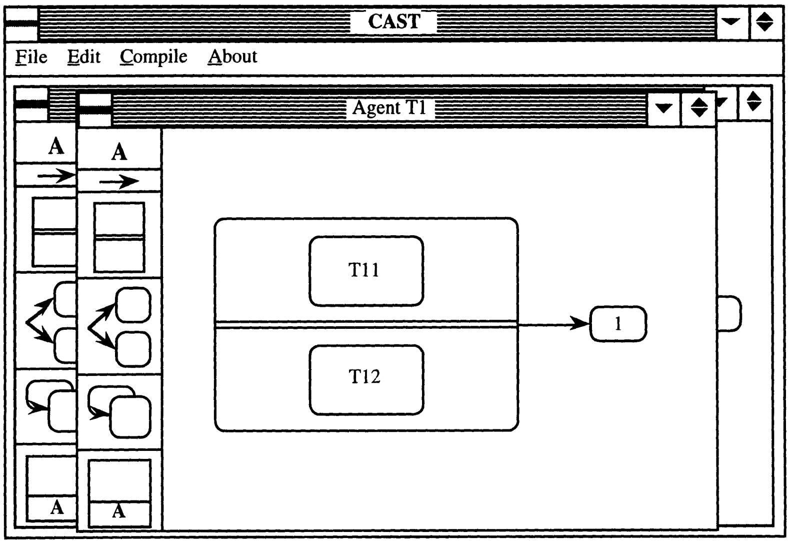

Fig. 5 - CAST user interface (global automaton for the task T1 of the process P)

The behaviour of *OP* consists in enumerating all the illegal situations and to verify that the current state of the observed process is correct. The observers can detect illegal situations, for example incorrect requests by the designers, objects that are in particular configurations (e.g., inconsistent objects cannot be made public). Incorrect interactions are also prevented in this way, when the objects are modified by simultaneous updates from the designers.

The verification formalism must ensure that all the information provided by the specification will be considered. Another way to verify the specification of a process is given by observers. This method consists in specifying for each process *P* an observer *OP* which behaviour consists in detecting illegal behaviours of *P*. These observers may be written by SCCS. The behaviour of *OP* consists in enumerating all the illegal situations and to verify that the current state of the observed process is right.

Actually we verify process models by the observer's technique. We plan to implement a verification mechanism in CAST based on a temporal logic. This verification mechanism consists in transforming the SCCS specifications into temporal logic predicates and to apply logical proof techniques in these predicates. SCCS specifications can be verified using linear temporal logic (Piard 1994).

5. IMPLEMENTATION OF PROCESS MODELS

The cooperative engineering processes are implemented using the object model in SHOOD (Nguyen 1992b). Beside the extensive use of knowledge representation concepts, they are based on the use of active rules for the implementation of the operations and for the scheduling of the tasks (Bounaas 1995).

The tasks are modeled in turn by scripts which include sets of active rules. The latter are in charge of the activation of the external basic operations in dedicated software, e.g., a geometric modeler. The reflexive nature of the object model used allows for the dynamic restructuring of the process templates, because they are modeled by classes, which structure and attributes can be modified on-line (Nguyen 1993).

The automaton described in section 4 was first implemented in ARGOS for a technology demonstrator (Nguyen 1996). ARGOS is an imperative language for the specification and verification of reactive systems. It is based on the hierarchical and parallel composition of automata. Unlike Statecharts, it has a sound and formally well defined semantics (Pnueli 1991). The full description of ARGOS is out of the scope of this paper. It is detailed extensively in (Jourdan 1994). Informally, automata are defined in ARGOS by states and transitions. The transitions are labelled by input and output events. Boolean event combinations trigger the transitions. Output events are produced when the transitions are fired. This allows the automata to communicate and to produce resulting events. The ARGOS compiler produces several output formats, which can be read by various specification and verification software, e.g., ESTEREL (Berry 1992). The automaton depicted in Figure 4 includes 46 states and 2425 transitions. The next step is to implement the process using CAST.

6. CONCLUSION

Many factors in different areas of computer science invite us to focus on formal techniques to specify a system, a software, etc. However, formal techniques are beyond the reach of the majority of the system's designers. Our domain of interest are reactive systems and communicating processes in concurrent engineering environments. We have chosen the SCCS algebra to specify them. Our experience in this domain shows that it is essential to have an approach or a tool which allows users to make the most out of the formal aspects of SCCS without the apparent complexity of mathematical problems. This reason lead us to develop CAST in order to help designers to model, specify and verify reactive systems and communicating processes. Using graphic tools is an advantage for design methods, especially if these tools are based on mathematical principles and allow us to verify the coherence of their results. CAST provides means to graphically analyse and specify consistently reactive systems and communicating processes.

The goal of the project SHOOD through CAST is to foster new developments that build on specification and verification techniques, together with research in knowledge sharing for advanced applications.

REFERENCES

Berry, G. and Gonthier, G. (1992) The ESTEREL synchronous programming language: design, semantics, implementation. *In: Science of computer programming.* 19(2).

Boudol, G. (1985) Le calcul MEIJE,Parallélisme, communication et synchronisation, *CNRS report.*

Bounaas, F. (1995) Using rules for object and schema evolution in an object-oriented system. *Proc. of 17th International Conference TOOLS'95.* Santa Barbara (USA).

Brown, D. R. et al (1992) Next-Cut: a second generation framework for concurrent engineering. *In: Entreprise Modeling and Integration.* C. Petrie (ed). McGraw-Hill.

Cutkosky, M. R. *et al.* (1993) PACT: an experiment in integrated concurrent engineering systems. *IEEE Computer.* 26 (1).

Dertouzos, M. *et al.* (1989) Made in America. *The MIT Press.* Cambridge MA.

Genesereth, M. R. (1992 a) An agent-based framework for software interportability. *Proc. DARPA Software technology conference* . Arlington (USA).

Genesereth, M. R. et al. (1992 b) Knowledge interchange format. Version 3.0 reference manual. *Computer Science Dept. Stanford University.* Tech. Report Logic-92-1.

Gero, J. S. (1993) Proceedings of the IFIP WG 5.2 Workshop on "*Formal Design Methods for Computer-Aided Design*". Tallinn (Estonia).

Guetari, R. (1995) Conception Orientee-Objet de Systèmes d'Information et de Decision. *PhD Thesis,* Université de Savoie (France).

Guetari, R. and Nguyen, G. T. (1996) A Class-Based Object-Oriented Model for Parallel Programming. The 1996 Parallel Object-Oriented Methods and Application, Santa Fe, New Mexico (USA).

Jourdan, M. (1994) Etude d'un environnement de programmation et de vérification des systèmes réactifs, multi-langages et multi-outils. *PhD thesis,* Université Joseph Fourier, Grenoble (France).

Maraninchi, F. (1992) Operational and compositional semantics of synchronous automaton compositions. *CONCUR. LNCS 630,* Springer Verlag.

McGuire, J. et al. (1994) SHADE: technology for knowledge-based collaborative engineering. *In: Journal of Concurrent Engineering.* 1 (2).

Milner, R. (1989) Communication and concurrency - Prentice Hall.

Newell, A. (1982) The knowledge level. *Artificial Intellignece,* 18 (1).

Nguyen, G. T. *et al.* (1991) Representing design objects. *In: Artificial Intelligence in Design'91.* Butterworth-Heinemann. J.S Gero (ed).

Nguyen, G. T. *et al.* (1992 a) Multiple object representations. *Proc. 20th ACM Computer Science Conference.* Kansas City (USA).

Nguyen, G. T. *et al.* (1992 b) SHOOD: a design object model. *In: Artificial Intelligence in Design'92.* Kluwer Academic Publ. J.S Gero (ed).

Nguyen, G. T. (1993) SHOOD: plate-forme pour la conception assistée. *In Ingénierie des systèmes d'information,* Hermès (ed) Vol. 1, N° 3.

Nguyen, G. T. (1995) A Reactive Object Model for Concurrent Engineering Design. *Research Report INRIA*

Nguyen, G. T. and Guetari, R. (1996) A Reactive Object Model for Concurrent Engineering Platforms. *Proc. 9th Intl. Symp. on Methodologies for Intelligent Systems.* Zakopane (Poland).

Patil, F. *et al.* (1992) The DARPA knowledge sharing effort: progress report. *Proc. 3rd International Conference on Principles of knowledge representation and reasonning.* Morgan-Kaufmann.

Paul, A. J. and Sobolewski, M. (1995) *Proceedings of the "Concurrent Engineering: a global perspective" '95 Conference.* A. J. Paul, M. Sobclewski (eds). Concurrent Technologies Corp. Washington D.C.

Piard, F. (1994) Spécification et Conception de Systèmes d'Information Dynamiques. *PhD Thesis,* Université de Savoie, France.

Pnueli, A. and Shalev, M. (1991) What is in a step: on the semantics of Statecharts. *In: Lecture Notes in Computer Science,* n° 526. Springer Verlag.

Tenenbaum, J. *et al.* (1992) Lessons from SHADE and PACT. *In Entreprise Modelling and Integration.* C. Petrie (ed). McGraw-Hill.