An Approach for Real-Time Applications Engineering

A. Attoui
LIMOS-ISIMA, BP 125 63173 Aubiere Cedex - France
Tel: 04 73 40 50 18, Fax: 04 73 40 50 01, email:ammar@sp.isima.fr

Abstract

Methods for complex and reactive systems engineering must produce specifications which must be *behaviorally expressive and rigorous* as well as *intuitive and well structured.* This paper presents an approach (supported by an environment called VALID) which allows specification, and system models generation. Models are fully executable and enabling ***full code synthesis***. *Valid code generation* is obtained by vertical refinement or formal transformations which guarantee its correctness and validity.

Keywords

Specification, Validation, Object-oriented, Reactive systems, Object-Oriented systems, Formal transformation.

1 INTRODUCTION

Formal methods and techniques have been proposed in order to ease the specifications design and verification. B, VDM... [HABR93, VIGD91, DAVI91, VDM91], and the STATECHARTS [HARE87] are some of the most important formal specification formalisms and techniques. These formalisms are different in nature, but their efficiency for detecting errors within specifications have *been well proved.* However, formal specifications remain **under-used** in the industry, for various reasons: necessary **proficiency in mathematics** and logic, **excessive formalism** that hardens communication between users.

For information, let us note that Statecharts are visual formalisms proposed by Harel [HARE97] for the specification of structural parts and the behavior of the studied systems. A Statechart specification is a state tree where leaves correspond to state diagrams (associated to finite state automatons). It allows a hierarchical description of a system

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35390-6_58

L. M. Camarinha-Matos et al. (eds.), *Intelligent Systems for Manufacturing*
© IFIP International Federation for Information Processing 1998

behavior with states and events. These events are broadcasted and instantaneously received: synchronous model.

As Petri nets, Statecharts have been expanded. Modecharts [Jaha88] are their most interesting extensions. They are dedicated to critical time systems. Theoretical basis of Modecharts is temporal logic. The basic principle is about extracting structured data present in Modechart-type descriptions of a system, an execution graph for the system usable for the validity verification of assertions that are explicitly described by the user using temporal logic. This execution graph includes time intervals [Min, Max] on events. In this model, the execution is assumed to be a sequence of partially ordered events. Thus, and unlike transitional systems, Modecharts do not bring modeling of real parallelism existing in the studied systems back to simple or pseudo-parallelism. This is indeed very heavy while reasoning about systems temporal properties.

As a conclusion, Statecharts and their extensions allow a hierarchical description of finite automatons, based on a graphical representation that specifies the visible behavior of a process. Statecharts are as easy to use as state-events diagrams. However, this language is so rich that it must be used carefully, because formal semantic is incomplete when it comes to describe limit cases. Besides, a lack of rigor can lead to hard to read or ambiguous diagrams. The synchronous model imposed by Statecharts, and all synchronous languages, can also produce causal paradoxes (events or effects that prevent their own cause from happening). Furthermore, Statecharts only offer possibilities to describe interactions between processes. To make possible the description of constraints or intrinsic properties of real-time systems, extensions like Modecharts, based on temporal logic, have been introduced, ***with all the difficulties of use it induces***.

To overcome some difficulties linked to these methods and formalisms, especially formal methods (necessary knowledge in mathematics and logic, excessive formalism that makes communication between users more difficult, lacks in methods and tools), we propose a global approach for reactive and concurrent systems engineering. Its main characteristics are the following:

- formal specifications are based on **rewriting logic theory**;
- the approach adopts **object-oriented** concepts;
- formal specifications are established through a **graphical browser** that masks the details of formalism, and allows VALID specifications to be fully executable and enables full **code synthesis**;
- **verification** of the behavior and the interactions in the studied system is possible with a simple **syntactical reduction** with constraint programming language PROLOG III; the studied system behavior simulation PROLOG III program is automatically issued by the graphical browser;
- automatic **valid code** generation of the application is obtained by **formal transformations**.

2. FORMAL MODULES (Description of an active object)

Our model is based on a formalism that is closed to the MAUDE language one [MESE90]. It takes into account the complexity of information processed within a

complex system, it includes a description of the various objects implicated, the actions that these objects may undergo, the temporal constraints and the traceability of information. **The model is independent of the target language**. It is based on the formal modules concept.

A formal module is a quadruplet (Σ, a, R, S). Σ is the set of symbols for the functions of the module. It allows the formal description of the static part of an object. Symbols can be simple (i.e. characters) or complex syntactical units. a is the set of structural axioms necessary to achieve the rewritings in a concurrent manner modulo the axioms.

The doublet (Σ, a) is called the signature of the object. R is a set of rewriting rules. It allows the formal description of the dynamical part of a system. A rule has the form [t] ==> [t'] where t and t' are terms constructed from Σ. The notation [t] is used to indicate that t represents an element of the class of terms modulo the axioms. We will use three structural axioms called ACI : associativity, commutativity and identity. A rule indicates that the *current state* of the agent corresponding to *the configuration* t, becomes a new state corresponding by *the configuration* t'.

2.1. The signature

For (Σ,a) we use a general formulation (figure 1) which is closed to the one proposed by [MESE90]. This general form of signature integrates different operators. Op < _: _ / _> is the constructor of agents. Op _ = _ allows to affect a value to an attribute. Op _, _ is the syntactical constructor of attribute lists. These lists are used for the designation of the attributes of an agent. Op _ _ is necessary for the construction of distributed configurations. It makes it possible the specification of any configuration. A configuration is composed of agents and messages. This operator has been declared modulo the ACI axioms. Thus, the order with which agents and messages are declared has no influence on the reduction process used further.

```
/*Alphabet of the system*/
Type  Agent, Attribute, Attributes, Msg, Configuration, Value, AgentId, ClassId,
AttributeId;
/*Hierarchies and structural relations between the agents*/
Subtype AgentId, ClassId, AttributeId < Value;
Subtype Attribute < Attributes;
Subtype Agent, Msg < Configuration;
/*Operators for constructing the words and the sentences of a formal module*/
Op < _: _ / _> : AgentId Value -> Object;
Op _ = _ : AttributeId Value -> Attribute;
Op _, _ : Attributes Attributes -> Attributes [ Assoc, Com, Id = Nul];
Op _ - : Configuration Configuration -> Configuration [Assoc, Com, Id =Nul].
```

Figure 1 The general signature of a formal module

The description of a real-time database application consists to instanciate the metatypes of Σ (agent, Attributes, Msg, ...). This instanciation is made automatically by a graphical editor from the graphical representation of an object (figure2).

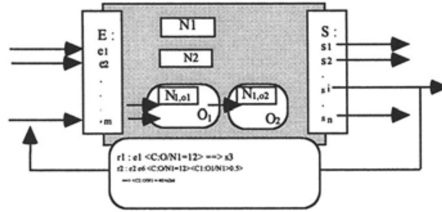

Figure 2 Graphical representation of a VALID object

2.2.Invoicing orders real-time database application

Figure 3a shows a VALID object-model diagram for an invoicing orders real-time database application with two main classes (Order and Product). AVALID object-model diagram specify the application (system) classes and their structural relationships as well as any Object-Oriented approach (Booch method, OMT or UML, OOSE). Numbers in the circles indicate multiplicity information (number instances of the class). If an asterisk is used, it indicates that the corresponding class can have unlimited instances. The edge between the two classes represents an association relationship (reference association). Ina general way, and like others structured analysis approaches (i.e. SA/RT,OMT, etc.), a VALID object-model diagram is hierarchical and features higraph encapsulation, which denotes a **strong composite class aggregation**. Directed edges represent relationships; an undirected edge is the same as a two-way directed edge. Directionality dictates the ability to reference instances: In figure 3a, a Product object cannot refer to the set of Order objects that refer to it. An association may have a name or a role.
According to the studied system nature, an association relationships expresses a **structural physical link** which must be established between objects of the two implicated classes. In an implementation stage, this link is supported by a **network connection** when we deal with distributed systems and applications, embedded systems and industrial process components supervision and control application, etc. In a general way, this kind of link concerns **physical systems**. For more classical **monolithic** software applications where application domain entities are supported by classical data structures (C++ classes, files, Relational databases tables, lists, etc.) an association relationship (reference) can be materialized by a sample **pointer**! or a set of pointers according to the association multiplicity.

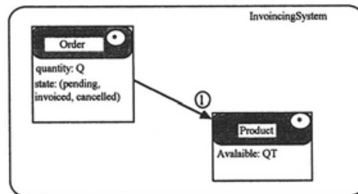

Figure 3a **Case 1** invoicing order VALID object-model diagram

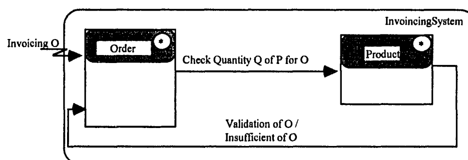

Figure 3b **Case 1** invoicing order VALID object communication diagram

Also, classes attributes are represented in VALID object-model diagram. For instance, `Quantity` indicates the quantity of Product `P` in order `O`. The attribute `state` of `Order` class can takes one of the three possible values : `pending,invoiced, cancelled`. At this stage, we have to decide if some attributes of `Order` and `Product` classes have to be made visible at this level (object or classes environment) or not? The response to this question is required for data protection, class interface protocol and future system architecture and control type (centralized or distributed) to be defined or constructed in the next step. Indeed, it is important to remember that an association or an aggregation denotes a physical link between the system components. Hidden class attributes are private and no direct access to them is possible. So we have to anticipate some explicit messages to manipulate them. As much as possible, it is better to privilege distributed architecture which ensure more protection, modularity, flexibility, reliability, object distribution and reuse. For our example, we have decided that each class attributes are hidden (private). Consequently, to invoice an order `O` which have reference to an ordered product `P`, we have (object `O`) to send a message (`Cheek Quantity Q of P for O`) to the `Product` object `P` and wait a response which must be (1) (`Validation of O`) if the ordered quantity is either less or equal to the quantity which is in stock, or (2) (`Insufficient of O`) in the other case. This decision will facilitates objects interaction and collaboration. In a centralized and monolithic architecture, Quantity access are supported by *method invocation* or *operation call.* So the above three messages will be translated to a Product class method signature : `Product::Check(int quantity)` which return `True` or `False` as response values. In a distributed version, these messages are *included in the application protocol.* It is important to note that this approach is well suited to support active objects.

If `Order` and `Product` class object attributes are made visible, this allows solutions where `Order` objects or any other user process can have direct access to `Available` attribute of the referred `Product` object and tests or updates it as in the following C++ instruction :

```
(itsProduct->Avalaible >= quantity)itsProduct->
                                    Avalaible =- quantity; (1)
```

As a consequent, data protection and consistency violation must occur. On the other hand, this solution concerns classical software application with **common global memory** computer architecture. Designer must take into account and manage all concurrency and

shared data accesses. In an attempt to overcome this problem early in specification stage, VALID allows visible attributes specification which is natural for many distributed and industrial applications and systems. *But access to these visible attributes is only done with a set of messages according to a Client-Server protocol.* An `Order` object (Client) has to request the value of `Available` attribute of `Product` object (Server). After, it can use this value or updates it and returns the new value to `Product` Object. As it will shown later in this paragraph, VALID generates automatically all necessary access messages for any visible attribute. So, VALID specification formalism contributes to reduce the use of instructions of (1) form in final system code. This approach is conform to Brinch-Hasen's concept of a monitor as a method of structuring complex systems which allows suitable proof rule.

2.3. Object communication and co-operation diagram

Because an object-model diagram describes classes and their structure, it appears to be concerned with static aspects only. However, if the studied system is of a limited complexity (like our example), this diagram can contain information that helps the dynamic behavior description of the system. Otherwise, we have to associate to each object-model diagram of the hierarchy a **communication diagram** (figure 3b).

Events and messages are used for interactions modeling between objects. ***A virtual channel concept*** is used to carry out a message generated by an object, and is queued, to be handed to the target server objects in its turn. The interface of each class indicates the input messages as well as the output messages. Virtual channels are used by VALID specification **translators** to implement object communication (interactions) with one of the available means (UNIX sockets, Ined sockets, common variables, FIFO Queues, Pipes, etc.) according to objects localization area and global system performances goal. Notice that messages are used to denote usual events generation or messages as well as object operation call. An object can invoke operations of an another object, causing it to carry out an appropriate method with an appropriate message arguments. The message `Check Quantity Q of P for O` can be translated into a message in a distributed system, or into a simple method invoking in a centralized implementation.

External events are generated from **environment** objects(InvocingSystem) : `Invoicing O` is an external solicitation with invoke the invoicing object O . They are handled by their target class objects (i.e. Order object O).

Internal events are collaboration and object synchronization **protocol**. For example, the message `Checking Quantity Q of P for O` is generated by an object of class `Order` when it receives the message `invoicing O` to verify that the necessary quantity `Q` can be satisfied before change it self state from "`pending`" to "`invoiced`".

In a general way, An object `O` can generate an event and send it to some other object `P`. If the event is of message type, it can also contain any parameter `Q`. To refer to the target or the server object, the client object can use the name directly if the server is one of its components in a composite or regular aggregation, or if the server and the client are of the same level and are components of the same composite or regular aggregation.

In any case, the reference denoted generically is:

```
<event name and information> (<parameters>) (<server>)(<client>).
```

Notice that the `<client>` information and the association between the two classes is used by the transformation process (translator) in the design stage to determine the type of the communication channel : private (simplex) and unidirectional if `<client>` is omitted, full-duplex and bi-directional if it is present.

2.4. Object behavior (The rewriting rules)
Actions of messages on an object are described with rules. An object (server) can receive messages from objects (clients) in the same level or in the upper levels of its hierarchy. An object can send messages to objects in the same level or in the lower levels of its hierarchy. A message can also be intercepted through a rule and routed to any level. A rule **signals the occurrence of a communication event** in which `n` messages and `m` objects are involved. All the objects participating in a rule are at the **same level (composite or regular aggregation)**. The general form of a rule is given by figure 4.

```
                    /*Syntax*/
M1M2..Mp<AG1: C1/ listeAt1>...<AGi: Ci/listAti>
  <Aj: Cj/ listAtj>...<Ak: Ck/ listAtk> =[T]=>
      <Aj: Cj/ listAtj>...<Ak: Ck/ listAtk>
<AGm: Cm/ listAtm>...<AGn: Cn/ listAtn>Mq ... Mr
```

Figure 4 Syntax and effects of a rewriting rule

Effects:
The messages M1M2..Mp are deleted after the execution of the rule.
The states of the agents Aj,..., Ak are modified.
Agents A1,..., Ai which appear only in the left part of the rule, are deleted.
New Agents AGm,...,AGn defined in the right part, are created.
New messages Mq, ..., Mr are created.
[T] is a temporal constraint on the rule transition (execution). It can take:
1: every (T, msg) : to each time interval T, send the message msg;
2: within (T, msg) : after the time T, send the message msg;
3: AT (T, msg) : at the time T, send the message msg;
4: before(T, msg) : before the time T is elapsed, send the message msg.

The originality of these operators is that they are easy to use in comparison with others formalisms, and adequately support a structured specification style. More details on rewriting logic and its applications can be found in [MESEG91]. When a rule is fired, messages and actions in the right part do not take effect until a stable situation has been reached. However, all rule triggers are constantly attentive, and generated messages reach their destinations in an atomic way (**become available in their reachable configurations which are supported by virtual channels in our approach**). The firing of object rules is a **multi-thread execution**: all satisfied rules are simultaneously activated.

Virtual channels must guarantee the atomicity of an event transport: events are handed to server objects and made visible in their private configuration. Moreover, we require all

parts of a rule to be fully executed before the rule becomes stable and can respond to other events or messages. Rules can be **periodic** or **aperiodic** (i.e periodic and aperiodic tasks). Aperiodic rules are messages or events handler. Periodic rules are cyclic tasks with a fixed activation period (`Every` temporal constraint). If a message is used in many rules in the same object, all these rules have simultaneously access to this message which is present in the private configuration of the object. If a message is used in several objects, a copy of the message is given to these objects and makes it available in their private configuration. As a consequence, when the left part of a rule contains several messages, it is more realistic to apply the "*rendez-vous*" concept (**synchronous rule**). In certain situations, the *rendez-vous* mechanism is not realistic, so virtual channels must offer an other alternative mechanism such as the **queued dated** events which become necessary to achieve the system coherence and stability. [ATTO97] gives more details on virtual channels implementation.

2.5. Invoicing orders real-time database application behavior

Our approach allows centralized and distributed control of systems activities. In the following section, we present a distributed control version of invoicing orders real-time database application. The next section presents a centralized version. It is obvious that concurrency is already inherent in an Object Oriented system because different object instances can exist and can operate simultaneously. Moreover, our model is a multiple-thread model, and different threads can execute simultaneously in different orthogonal components of same object (i.e. an object Order of identifier O1 and an other object order of identifier O2 can execute simultaneously). It is obvious that this approach is not suited to real-time database applications, where the number of classes instances is important. On the other hand, it is well adapted to real-time systems modeling such as manufacturing systems, real-time embedded systems, hardware computer systems, etc. However, we present here the distributed version of invoicing orders real-time application, just to show how our approach can take this problematic into account . The centralised version of this application is more realistic. During a VALID specification session, the user must take into account the following rule :

For each VALID object-model diagram (at any level), we have to decide between two control and synchronization approach:

- **Distributed control**: diagram components are autonomous and the collaboration protocol highlighted by the communication diagram (control and data streams and components interface) is sufficient to implement the system behavior. In this case, components **are active and reactive objects** (i.e. hardware chips, manufacturing station or module, network node, etc.)and does not need any supervision and global control. In normal conditions, each component instance (class instance) must be active (i.e. executed by a separate thread for software systems).
- **Centralized control**: some or all diagram components **are passive objects or resources** (data representation and structures: classes, lists, tables, or physical resources: sensors, pallet, etc.) and necessitate external and global control to use and manipulate their content.

2.5.1. Distributed control

It is important to remember that a virtual channel mechanism which supports the configuration concept in VALID is used for interobject communication as well as complex internal objects interactions and behavior specification. It is different from the broadcasting mechanism which is unrealistic for many systems. Figure 5a gives the `Orders` class objects structure and behavior, and figure 5bgives the behavior of `Product` class objects.

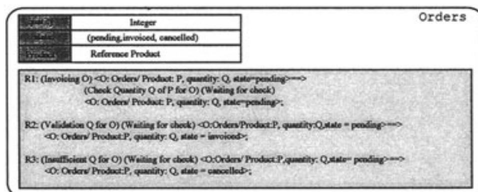

Figure 5a Orders class objects behavior and structure description

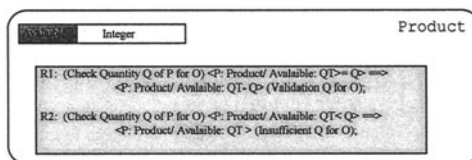

Figure 5b Product class objects behavior and structure description

The rule `R1` of an `Orders` class object `O` is fired if and only if the object `O` receives the message (`Invoicing O`) and its state is in "`pending`". The result is the generation of the two messages : `(CheckQuantity Q of P for Q)` and `(Waiting check)`. The second message or event is an internal service event (i.e. flag) only used to indicates that the object O have to take into account asynchronization event and several alternative responses for the message `(Check Quantity Q of P for Q)`. This message stays available in the private configuration of object `O` until it receives a response. However, in this case, it is more realistic to associate to rules R2 and R3 a temporal constrain (timer or dog watcher)in order to avoid object O to wait infinitely the Product object response:

```
R2 (Validation Q for O) (Waiting for check)
<O: Order/ Product: P, quantity: Q, state="pending">
=[before(20, (TimeOut for O)]=>
<O: Order/ Product: P, quantity: Q, state="invoiced">
```

The temporal constraint indicates that the rule has to be fired before 20 time units from the moment one of the two messages of the left part becomes available, which is the case for the message: `(Waiting for check`. If the message `(Validation Q for O)` arrives in time the attribute `state` is set to `"invoiced"` and the two messages of the left part of the rule R2 are deleted from the object configuration.

As a consequence of this, we have to create another rule to eliminate (i.e. updating the corresponding flag in the design stage) the service message :

```
(TimeOut for O) (Waiting for check)
<O: Order/ Product: P, quantity: Q, state="pending">  ==>
<O: Order/ Product: P, quantity: Q, state="pending">
```

2.5.2. Centralized control

The previous version of invoicing orders real-time database application uses **reactive and co-operative objects**. In an implementation stage, they can be supported by one machine as well as by several interconnected machines. Figure 6 gives a centralized version of the same application. This version is more realistic according to the application type. In this case, rewriting rules must be associated to the global and abstract class: `InvoicingSystem`.

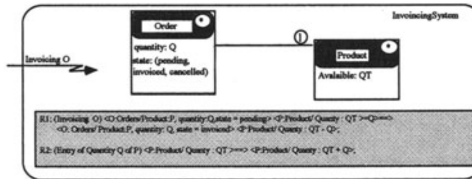

Figure 6a Centralized invoicing orders real-time database application version

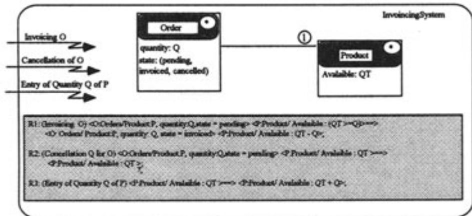

Figure 6b Centralized invoicing orders real-time database application version

In this version `Order` and `Product` object are considered as `InvoicingSystem` objects or resources. The rule R1 is a **synchronous** rule. The execution (firing) of this rule is done in an atomic way. Objects which appear in the left part must be processed within a **critical section** (mutual exclusion). Any translation of this kind of rules into a target language must take this characteristic into account. Finally, figure 6bgives the centralized version of our example.

3 CONCLUSION

The analysis of this study case with VALID has followed the two main steeps of VALID specification methodology:

- **System architecture description** : Object-model diagram highlights the system object classes (objects) and their structural relationships. In this steep, it was important to decide which attributes of the two classes (`Orders` and `Product`) to be made visible or not at the system level. For this attributes, VALID generates the necessary rules to manipulate their contents and guaranty their consistency. The Object communication and co-operation diagram highlights objects interfaces (data or control input/output) and classes interconnections. Relationships, Input/output and attributes are used by VALID translators to grantee valid code generation. Also, static properties such as objects or data consistency and object class relationship associations (references, etc.) can be easily handled by using specific rewriting rules.
- **System behavior** : Rules are event or messages handlers. It was important to chose between centralized or distributed control. Indeed, in classical applications, we specify in a distributed manner and in the design steep we choose between centralized or distributed implementation. The main program of an application constitutes the objects control and supervision. VALID is a more general specification environment which take into account classical and computer applications as wheel as industrial and automated systems analysis and specification. Prolog III model generated by VALID allows the verification of the application **protocol or interactions** properties: deadlock, application stability, reliability and fault tolerance property according to external or internal exceptions and errors arising, and any dynamic aspect inherent to system behavior. [ATTO96, ATTO97] give some system dynamic simulation and temporal properties verification examples. Simulation models (Prolog III program) and final program code generation in a target language (CC++, ADA, etc.) is obtained by vertical refinement techniques [ATTO97a] which has the advantage to generate valid program from high level specifications. A correct program generation is achieved by transformations according to VALID pivot textual formalism and the target language semantics.

4 REFERENCES

ATTOUI A. and SCHNEIDER M. (1996) "*A Formal Approach for the Specification and the Behavior Validation of Real-Time Systems Based on Rewriting Logic*", Real-Time Systems Journal, Vol 10, N¡1, January.

ATTOUI A. (1997),"*An Environment Based on Rewriting Logic for Parallel Systems Formal Specification and Prototyping*", Journal of Systems Architecture, Vol 44.

ATTOUI A.,HASBANI A. (1997a), "*Reactive Systems Developing by Formal Specifications Transformation*", 8th International Conference and Workshop on Database and Expert Systems Applications, September 1-5,Toulouse.

JAHANIAN F. and STUART D.A. (1988), " *A method for verifying properties of Modechart specifications*", IEEE Real-Time Systems Symposium, Huntsville, Alabama, December.

LIGHTFOOT D. (1991) "Formal Specification Using Z". The Macmillan Press.

HABRIAL H. (1993), "*Introduction ˆ la specification*", Editions Masson, Paris.
HAREL D. (1987), "*Statecharts: a visual formalism for complex systems*", Science of Computer Programming, Vol. 8, N¡3, June.
HAREL D. and Gery Ecran (1997), "*Executable Object Modeling with Statecharts*", Computer, Vol. 30, N¡7, July.
MESEGUER J. (1990)"*A Logical Theory of Concurrent Objects*". in Proceedings Concur'90 Conference, Springer Verlag, Amsterdam, August .
VIGDER M. (1991), "*Using VDM within Object Oriented Framework*". in Proceedings VDM91 Formal Software Development Methods, Noordwijkerhout, the Netherlands, 21-25 Oct.
VIGDER M. (1991) "*Using LOTOS in a Design Environment*". in Proceedings FORTE'91, Sydney, 12-22, Nov.