Real-Time Systems, 8, 117–172 (1995)

Tools for Specifying Real-Time Systems*

GIACOMO BUCCI
Department of Systems and Informatics, Faculty of Engineering, University of Florence, Florence, Italy.

MAURIZIO CAMPANAI
CESVIT/CQ_ware, Centro per la Qualità del Software, Florence, Italy

PAOLO NESI nesi@ingfi1.ing.unifi.it
Department of Systems and Informatics, Faculty of Engineering, University of Florence, Florence, Italy.

Abstract. Tools for formally specifying software for real-time systems have strongly improved their capabilities in recent years. At present, tools have the potential for improving software quality as well as engineers' productivity. Many tools have grown out of languages and methodologies proposed in the early 1970s. In this paper, the evolution and the state of the art of tools for real-time software specification is reported, by analyzing their development over the last 20 years. Specification techniques are classified as operational, descriptive or dual if they have both operational and descriptive capabilities. For each technique reviewed three different aspects are analyzed, that is, power of formalism, tool completeness, and low-level characteristics. The analysis is carried out in a comparative manner; a synthetic comparison is presented in the final discussion where the trend of technology improvement is also analyzed.

1. Introduction

In recent years, several techniques for formal specification of real-time systems have been proposed; a large number of tools — ranging from research prototypes to marketed software packages — supporting these specification techniques have also been introduced. The growing interest for specification tools can be explained by considering that they have the potential for improving software quality as well as engineers' productivity. Furthermore, their use may be the only practical way to guarantee that certain quality factors (such as safeness, consistency, timeliness, etc.), which are mandatory for real-time systems, are achieved.

This paper contains a historical review above the tools for the specification of real-time systems, taking into account their evolution in the last 20 years. To this end, a number of well-known proposals are examined and criticized in the light of the classification criteria described in the sequel. The choice of classification criteria has been one of the major concerns of this paper. In fact, space limitations do not allow going into details for each tool under examination. Furthermore, the subject matter is far away from being stable and settled and so we may have overlooked some relevant issues. As a result, we do not claim that the proposed taxonomy is an exhaustive method for classifying real-time specification techniques. However, we believe that this paper provides a reasonable

* This work was partially supported by the Italian Research Council, CNR (Consiglio Nazionale delle Ricerche), n. 93.01865.CT12.

picture of their evolution as well as an indication on future developments and research issues.

A popular method for classifying software specification techniques is based on the degree of formality used. *Formal* techniques are based on mathematics, and (pure) *informal* techniques on natural languages. The former are generally preferred, because the latter tend to be incomplete and inconsistent (Meyer, 1985), (Stankovic, 1988), (Levi and Agrawala, 1990), (Stankovic and Ramamritham, 1992). The formalism can cover both syntax and semantics of a technique or only a part of them.

Another method for classifying software specification techniques is based on the extent to which they are *descriptive*, *operational* or *dual* (that is a mixture of descriptive and operational). *Operational* techniques are those which are defined in terms of states and transitions; therefore, they are intrinsically executable. *Descriptive* techniques are based on mathematical notations (axioms, clauses, etc.) and produce precise, rigorous specifications, giving an abstract view of the state space by means of algebraic or logic equations. These can be automatically processed for verifying the completeness and the consistency of the specification, by proving properties by means of automatic tools. *Dual* techniques tend to integrate both descriptive and operational capabilities, allowing the formal specification by means of clauses or other mathematical formalisms as well as the execution of specifications based on state diagrams or Petri nets. For dual techniques, the main problem is the formal relationship between operational and descriptive notations, which should be interchangeable.

In the literature, there are many other classifications according to which tools are divided in process-, data-, control-, and object-oriented (Dorfman, 1990) or in model-, and property-oriented approaches (Wing, 1990a), (Hall, 1990). In (Zave, 1990), Zave has made a classification by considering the degree of formalism with respect the degree of descriptiveness/operationality. This has resulted in a plot having in the abscissa the formal-informal range, and in the ordinate the descriptive-operational range.

In this paper, a somewhat different approach is taken. The classification is based on the distinction between operational, descriptive and dual techniques; however, for each technique reviewed three different aspects are analyzed, that is, *power of formalism*, *tool completeness*, and *low-level characteristics*.

In the light of this approach, languages and tools for reactive system specification are historically surveyed. It should be noted that a classification based on the distinction between operational, descriptive and dual aspects quite mirror the historical evolution of the subject matter. The remaining part of this section gives a brief account of the factors which must be considered in assessing tool capabilities. The review is made by presenting several examples in order to better explain the main characteristics of each technique with respect to these factors.

In evaluating the *power of formalism* three different aspects must be considered: the structural, the functional and the behavioral aspect (Harel, 1988), (Wing, 1990a). The *structural* aspect refers to the system decomposition into sub-systems. The *functional* aspect has to do with the transformational activities (on data) performed by individual software components. The *behavioral* aspect (i.e., the system dynamics) refers to the

system reaction to external stimuli and internally generated events, either synchronously or asynchronously. The systems in which the behavioral aspect is relevant are usually denoted as reactive; real-time systems belong to this category. Descriptive methods usually fail in modeling structural and functional aspects, but they are suitable for describing system behavior. Operational methods are intrinsically suitable for modeling system behavior in detail, even if they lack in mathematical foundation for describing system behavior at the needed level of abstraction in order to allow validation (i.e., the proving of a required property) without simulation.

Since this paper is focused on tools for real-time systems specification, a particular attention is devoted to identifying the expressivity of tools in modeling the *temporal constraints* (timeout, deadline, etc.) (Stankovic, 1988). At a high level, a formalism can deal with time either in an *explicit* or *implicit* manner. In the first case, the language allows the representation of time through variables which provide an exact time measure. Explicit timing constraints can be expressed in relative or absolute form. When time is expressed in a relative manner, time durations and deadlines are given in time units. In this case, the relationship between these time units and the absolute measure of time expressed in seconds (or milliseconds) is not clear. However, the validation of specifications becomes almost hardware independent. When time is expressed in absolute form, time durations and deadlines are directly given in seconds or milliseconds (i.e., the absolute time of the clock) and therefore the meeting of timing constraints depends on the context (machine type, number of processes, workload, etc.). When time management is implicit, the formalism is able to represent the temporal ordering of events without reporting any quantification on time intervals (i.e., in state machines, and in languages enriched with operators like *next* and *previous*). When time is treated implicitly, the possibility of its exact measure is usually lost.

A tool for specifying real-time system should guarantee both correctness and completeness of the formal specification, as well as the satisfaction of system behavior with respect to both the timing constraints defined and the high-level behavioral descriptions. The *verification* of correctness and completeness is usually performed statically by controlling the syntax and semantics of the model without executing the specification. The system *validation* consists in controlling the conditions of liveness (i.e., absence of deadlock), safety, and the meeting of timing constraints (e.g., deadline, timeout, etc.). It is usually performed statically in descriptive approaches (i.e., by proving properties) and dynamically in operational approaches (i.e., by simulation). The capability of the method for verifying and validating the system specification (by means of mathematical techniques or simulations) must be analyzed in order to establish if the tools are capable of guaranteeing that the specification produced exactly matches the behavior of the system under development, with safety, without deadlocks and by meeting all timing constraints.

Please note that the two terms verification and validation do not always receive the above meaning (Thayer and Dorfman, 1992). For instance, in (Thayer and Dorfman, 1992) the most frequently used definitions for *verification* are reported, while the term validation is mentioned as "final verification".

A specification model may be *executable* or not. Executability is, by definition, a prerogative of operational approaches. On the other hand, some descriptive specifications

are also executable. Executable means that the model is focused on defining the possible evolutions of the system state domain, rater than describing what the system should perform.

A model can provide support for defining strategies for *recovering from failures*, such as timeout, overflow, divide by zero, unmet deadline, etc. as well as for managing *external exceptions*. These features are usually available in operational approaches. Since there are some difficulties to guarantee that timing constraints are met, simulation is the way for detecting where the recovering paths must be defined.

Referring to *tool completeness*, an *integrated specification environment* is mandatory. To this end, a formal language must be endowed with a set of features, helping the user in its work. A very relevant feature is the availability of a well-defined *methodology*. In fact, a tool supported by a methodology is easier to be learned and used, while the quality of specification improves, and becomes more stable, irrespective of user's experience. Moreover, a tool should support the analyst in all phases of *software life-cycle*. Operational approaches are mainly based on the design aspect of the problem, while descriptive are better ranked for supporting the analysis phase.

The user interface is also very important. In recent years, graphics user's interfaces and *visual* languages have highly improved user's productivity with respect to *textual* interfaces. From this point of view, the operational approaches are favoured since they are intrinsically endowed with a visual notation, while the definition of visual language supporting the syntax of descriptive approaches is a more difficult task.

Another important factor is the presence of an automatic *generator of documentation* and of a *code generator* for classical high-level languages, like C, C++, ADA, etc. Note that several tools provide the support for generating a skeleton of the final code; in these cases, the verification and validation phases are usually performed without considering the final version of the code. Therefore, its final behavior may be unpredictable.

A further ingredient for improving users' productivity is the support for *simulation*. The simulator may either execute the generated code or interpret the specification itself. The first case is typical of the operational approach, whereas the second corresponds to the descriptive approach. Of course, the first method produces more trustable results, since the specification execution is based on the same code as that which will be executed at run-time. Simulation is usually performed by controlling system behavior with respect to manually or automatically generated test patterns. Of course, the latter are to be preferred; however, automatically generated test patterns should be used judiciously, since they may be affected by vices. The analyst should also check if test patterns provide a sufficient coverage.

In order to guarantee the possibility of *prototype generation*, a tool must provide support for *validating partially defined specifications*, otherwise the prototype might produce an unpredictable behavior.

Reuse of old specifications has become an important issue in software engineering. Many formal methods lead to specifications in which single components are too much coupled to be easily separated for reuse. Recently produced tools incorporate object-oriented concepts, since the object-oriented paradigm provides a number of mecha-

nisms for reuse (e.g., inheritance, instantiation, etc.) (Nierstrasz, 1989), (Booch, 1986), (Meyer, 1988).

With the term *low-level characteristics*, we refer to the underlying assumptions and to the basic environment on which the specified software is to be run.

In many approaches, the system under specification is modeled as a set of communicating sub-systems. The communications among these sub-systems can be synchronous or asynchronous. *Synchronous* mechanisms are more predictable, but also more sensitive to deadlocks; on the contrary, *asynchronous communications* are less predictable and less sensitive to deadlocks. In general, synchronous mechanisms are more suitable for specifying real-time systems — i.e., systems in which predictability is the first goal. Situations leading to deadlocks can be detected during the validation phase.

In order to guarantee the system predictability several *restrictions* are usually imposed. Most of them are devoted to constrain the possibility of changing the operating machine conditions. For example, no dynamic creation of processes or data, no dynamic process priority changes, and no recursion or unbounded loop definition are allowed. In the object-oriented specification tools, the absence of dynamic inheritance is usually supposed, and thus the possibility of defining polymorphic class hierarchies (Nierstrasz, 1989), (Booch, 1986).

Most of the tools proposed in the literature are supported by a specific real-time kernel, which includes a scheduler — e.g., (Sha and Goodenough, 1990), (Forin, 1992), (Liu and Layland, 1973), (Tokuda, Nakajima and Rao, 1990). Others approaches generate a code for platforms in which a real-time operating system is available. A choice among the several solutions is quite difficult. On the other hand, the specification tools take usually into account the features of low-level support in their semantics.

As already mentioned, the survey is focused on the historical evolution of tools and is organized as follows. Section 2 contains a short summary of early supports for modeling communicating concurrent processes. This is useful since in many cases they are the foundation for approaches surveyed in this paper. In Section 3 operational methods are examined, while descriptive methods are discussed in Section 4. Dual methods are treated in Section 5. The impact of CASE (Computer Aid Software Engineering) technology on tools is discussed in this paper with the comments related to the different techniques. In Section 6, the findings of our analysis are synthesized and some conclusions are drawn.

We are aware that many interesting languages, techniques and tools have not been considered. This is not to be viewed as a negative aspect, but rather as a necessity due to space limitations. Moreover, since this paper is mainly focused on tools rather than on languages, the latter are reported when their quotation is needed to explain the historical evolution of tools. A complementary survey focused on real-time languages can be found in (Stoyenko, 1992).

2. Mathematical Supports

In this section, the most frequently used mathematical supports for reasoning on communicating concurrent processes are briefly discussed. In the late 1970s, Hoare, with his work on CSP (Communicating Sequential Processes) (Hoare, 1978), (Hoare, 1985), and Milner, with his work on CCS (Calculus of Communicating Systems) (Milner, 1980), have posed the bases for the verification and validation of concurrent systems. The relationships among these two models have been discussed in (Brookes, 1983). Until (Hoare, 1978) several methods for specifying communicating sequential processes were widely used, including semaphores (Dijkstra, 1968), conditional critical regions (Hoare, 1972), monitors and queues (concurrent Pascal) (Brinch-Hansen, 1975), etc. As observed in (Hoare, 1978), 'most of these are demonstrably adequate for their purpose, but there is no widely recognized criterion for choosing between them'. This consideration led Hoare to attempt to find a single simple solution to all those problems. In the light of the subsequent evolution, CSP is considered as a first rigorous approach to the specification of concurrent systems.

The mathematical bases of CSP have been widely used for defining and analyzing concurrent systems regarded as processes communicating via *channels* (Hoare, 1985). For this reason, the CSP model is denoted as process-oriented, and each process is modeled as a sequential machine. The communication mechanism is completely synchronous — i.e., the transmitter/receiver is blocked until the receiver/transmitter is ready to perform the communication. In the CSP notation, sending a message e on a channel c is denoted by $c!e$, while receiving a message e from a channel c is denoted by $c?e$. This syntax and communication model have been frequently used for defining programming languages (e.g., Occam) and specification tools. In CSP model constructs for modeling parallel ($\|$), sequential ($\gg$), and interleaved ($\|\|$) executions of processes are also defined (Hoare, 1985).

Given its popularity, the original CSP model (Hoare, 1978) has been expanded in many ways, resulting in a set of models of increasing complexity: the Counter Model, the Trace Model, the Divergence Model, the Readiness Model, and the Failure Model (Moore, 1990), (Olderog and Hoare, 1986), (Hoare, 1981), (Hoare, 1985), (Misra and Chandy, 1981). The Failure Model can be profitably used for reasoning about the safety and liveness conditions of the system under specification, even in the presence of divergent models (i.e., having an infinite number of states) and non-deterministic processes (Barringer, 1985), (Hoare, 1985). The Trace Model can be used to analyze the history of events on the system channels, and for verifying if the system satisfies abstract descriptions of system behavior. For these reasons, CSP is an appropriate basis for both operational and descriptive approaches.

The CSP model does not comprise the concept of time and, thus, the system validation does not take into account timing constraints. For these reasons during the 1980s many extensions have been proposed for adding time support — e.g., CSP-R (Koymans, et. al, 1985) (where time managing is added by means of *WAIT t* instruction), Timed CSP (Reed and Roscoe, 1986) (where time managing is added by means of

the special function *delay()*), CSR (Communicating Shared Resources) (Gerber and Lee, 1989)and in the CRSM (Communicating Real-time State Machines) (Shaw, 1992) (where time is added by means of time bounds on executions and inputs/outputs), etc.

The syntax and semantics of CCS are based on the concept of *observation equivalence* between programs: a program is specified by describing its observation equivalent class which corresponds to the description of its behavior. This is given by means of a mathematical formalism in which variables, behavior-identifiers and expressions are defined. Behavior-identifiers are used in behavior expressions where the actions performed by the system are described. This makes the CCS model quite operational as pointed out in (Milner, 1980) and (Olderog and Hoare, 1983). This model is based on an asynchronous communication mechanism. The CCS model provided the ground for several models proposed in the late 1980s — e.g., (Bolognesi and Brinksma, 1987).

It should be noted that, the fact that the CSP model is strictly synchronous is not a limitation. In fact, by means of synchronous communicating state machines, asynchronous communications can also be defined. This is done through buffers of infinite capacity which are modeled as state machines as in (Shaw, 1992). In a similar manner, synchronous communications 1:1 (one sender and one receiver) can be expanded to 1:N communications (one sender and N receivers).

3. Operational Approaches

Operational approaches describe the system by means of an executable model. The model can be mathematically verified (for consistency and completeness) by using static analysis, and validated by executing the model (i.e., simulation). Though operational techniques were already introduced in the 1970s (Alford, 1977), it was not after the paper by Zave (Zave, 1982) that they have attracted large research attention. The most innovative aspect of Zave's paper was the embodiment of the operational approach into a programming language named PAISLey.

Operational approaches can be divided in two categories.

The first category comprises languages and methods which are usually based on transition-oriented models, such as state machines (Bavel, 1983) or Petri nets (Reisig, 1985), that is, models naturally oriented towards the description of system behavior.

The second category includes methods which are based on abstract notations especially suitable for supporting the system analysis and design (system decomposition/composition). In these cases, the notations are mainly oriented towards the description of system structure and/or functionality. For this reason, these notations are usually associated with guidelines for system analysis/design and are regarded as methodologies. However, most of them do not model system behavior and, thus, cannot be directly used for system simulation and specification execution. Moreover, since these methods have been developed by starting from visual notations, they lack in formalism and are usually considered as semiformal.

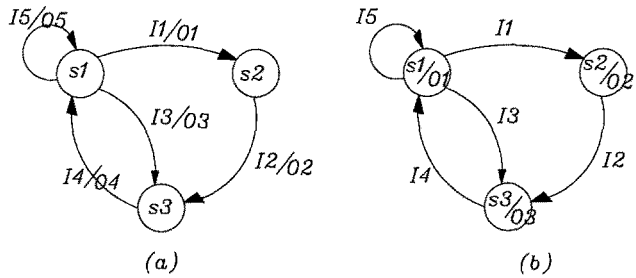

Figure 1. Example of FSM: (a) Mealy model, (b) Moore model, where Ii are input names, Oi outputs and Si names of states. In this example, the two models do not represent the same behavior. On the contrary, the Mealy and Moore models are interchangeable by means of simple rules.

3.1. Operational Approaches Based on State Machines

The basic theory of Finite State Machines (FSM) and automata dates back to the 1950s (Moore, 1956), (Mealy, 1955), (Booth, 1967), (Bavel, 1983). Since state machines are intrinsically operational, they have been used as a basis for several operational models.

The classical FSM models (by Mealy and Moore) are suitable for the specification of system behavior. Referring to Fig.1, outputs are produced as a function of the state of the FSM (i.e., Moore model) or of the state and machine inputs (i.e., Mealy model). FSMs can be represented by using two different notations: state transition diagrams (see Fig.1) and state transition matrices (Bavel, 1983).

The definition of FSM can be verified in order to identify its correctness — i.e., the reachability of the states, etc. In addition, the consistency and the congruence of a system description given in term of FSM can be verified by using several mechanisms as in (Jaffe, et. al, 1991). In a system defined as set of communicating state machines the number of states depends on the Cartesian product of the state domain of each machine; therefore, the number of states grows very quickly, and so the complexity of system verification. Moreover, in the presence of a communicating FSM there exists the possibility of deadlock and starvation (Hoare, 1985), (Sifakis, 1989). For this reason, mathematical supports, such as CSP and CCS are useful for reasoning about system and process (i.e., in this case FSM) liveness.

The classical model for FSM is unsuitable to represent the structural and functional aspects of the system under specification. To be profitably adopted as a tool for specifying real-time systems, the FSM model must provide support for describing temporal dependencies and constraints (e.g., timeout, deadline, etc.) among control flows. For these reasons, several extensions have been proposed in the literature.

Moreover, the classical FSM model is unsuitable when the number of states is very high. This problem can be partially avoided by decomposing the system into smaller communicating FSMs. The result is that the complexity of the system corresponds to the Cartesian product the of single machines state domains. Another possibility is to represent the state diagram in a more concise way with respect to the classical notation.

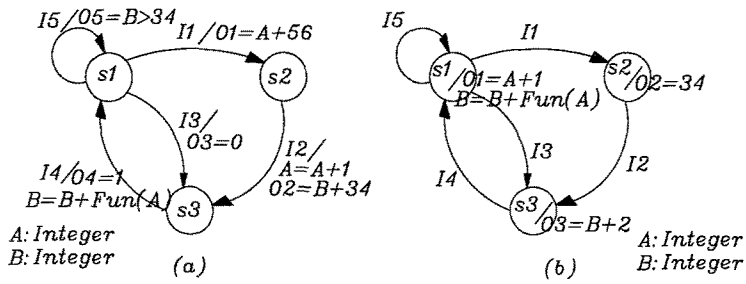

Figure 2. Example of extended FSMs: (a) Mealy model, (b) Moore model. Presence of expressions and of auxiliary variables — i.e., *A, B* of type *Integer*.

Classical FSMs are obviously unsuitable (being called Finite State Machines) to represent systems with a number of infinite states. Some extensions have been defined to give major expressivity to the graphical notation and to allow the definition of auxiliary variables (see Fig.2).

In FSM models the timing relationships among different events are implicitly defined by means of state diagrams. On the other hand, in several cases is not always possible to predict the state trajectory and, thus, the ordering of events. For this reason, an explicit model of time has been added to the FSM models in order to cope with the problems of real-time system specification.

One of the first integrated operational tools for software specification was the SREM (Software Requirement Engineering Methodology) (Alford, 1977). This was sponsored by the Ballistic Missile Defence Advanced Technology Center (BMDATC) in 1973. SREM is based on RSL (Requirement Statement Language) which supports both model-oriented and property-oriented styles of programming, and on REVS (Requirements Engineering Validation System) which is able to control the consistency and completeness of the specification. A system in SREM is decomposed in sub-functions. In SREM, the elementary function is modeled as an extended finite state machine. The state diagram is given in terms of the so-called R-nets. An R-net represents the system evolution starting from a state, by means of reading inputs, producing outputs, iterations, and showing in this way the set of possible next states. Since a finite state machine can have only a single active state, there is only an active R-net. Timing constraints are also defined into the R-nets. It should be noted that, the decomposition of the system in R-nets allows the specification of a distinct behavior for each system state without giving a global view of the system.

SREM, as an operational approach, has been widely used for more than a decade (Alford, 1985), (Scheffer, Stone and Rzepka, 1985). In parallel, several other operational methods sustained by mathematical formalisms were introduced and publicized. In the following subsections, four of them will be discussed — i.e., PAISLey (Zave, 1982), SDL (Rockstrom and Saracco, 1982), Esterel (Berry and Cosserat, 1985), and Statecharts (Harel, 1987).

3.1.1. PAISLey

PAISLey (Process-oriented Applicative and Interpretable Specification Language) was introduced in 1982. It is an operational specification model for defining embedded real-time systems (Zave, 1982), (Zave, 1984), (Zave and Schell 1986). The system under specification is decomposed in processes which communicate asynchronously. Although the communications are asynchronous, the model presents several methods for process synchronization. Each process is equivalent to an extended state machine. State machines are defined by means of a functional language. On the state domain of the system under specification, union, Cartesian product, and other operations are possible. In PAISLey, the external environment is also modeled to avoid misunderstandings in validating the specification. The number of processes, data structures and state domains are finite; these assumptions augment the predictability of system behavior. For example (Zave, 1982), a system decomposed in four processes is declared by means of their initial state:

(terminal_1_cycle[blank_display],
terminal_2_cycle[blank_display],
terminal_3_cycle[blank_display],
database_cycle[initial_database]
);

where *terminal_3_cycle*, etc.. are transition functions, while *blank_display*, *initial_database* are values of variables. For each process the domain range is defined, such as:

terminal_1_cycle: DISPLAY $\rightarrow$ *DISPLAY;*
. . .
. . .
database_cycle: $\rightarrow$ *DISPLAY;*

The single process (i.e., transition function) is defined on the basis of other functions, e.g.:

terminal_1_cycle[d] = display[display_and_transact[(d,think_of_request[d])]];

represents the internal structure of *terminal_1_cycle* functions.

Communications among processes are obtained from selected combinations between three modalities: $x\_$, $xm\_$, and $xr\_$, obtaining four type of mechanisms: (i) $(x\_, xm\_)$ blocked synchronous with mutual exclusion, (ii) $(x\_, x\_)$ blocked synchronous, (iii) $(xm\_, xr\_)$ mutual exclusion asynchronous, (iv) $(x\_, xr\_)$ simply asynchronous.

The method adopted for modeling temporal characteristics of the system is based on the theory of random variables. Therefore, PAISLey is primarily operational, but its timing constraints are mathematical (Zave, 1990). Timing constraints can be associated with the execution of state transitions, and are given by means of (i) lower and upper bounds, or (ii) distributions. For this reason, the static verification of time requirements could be only verified by means of a statistical reasoning. As a consequence, timing constraints

are checked in the phase of simulation, during which time failures are recorded. The final system validation is obtained by simulation, being the simulator an interpreter of the language.

3.1.2. SDL

SDL (Specification and Description Language) is an operational language belonging to the standard FDT (Formal Description Techniques) defined in 1982 within ISO (International Organization of Standardization) (ISO TC97/SC16/WG1) and CCITT (Comité Consultatif International Télégraphique et Téléphonique) for the specification of open distributed systems (Rockstrom and Saracco, 1982). In particular, the subgroups B and C have analyzed the descriptive models which combine the concepts of finite state machines with high-level languages, e.g., Pascal. SDL (Rockstrom and Saracco, 1982), (Sarraco and Tilanus, 1987), Estelle (Budkowski and Dembinski, 1987), and LOTOS (Bolognesi and Brinksma, 1987) (see Section 4.1.2) belong to this category of specification languages. SDL provides both visual and textual representations of its syntax. The textual representation extends Pascal according to the ISO draft.

In SDL, the system under specification is regarded as a block which can be decomposed in sub-blocks, thus modeling the structural aspects of the system (see Fig.3a). Blocks communicate asynchronously by means of strongly typed channels. The language provides support for defining new types of messages modeled as ADTs (Abstract Data Types) (Guttag, 1977), (Guttag and Horning, 1978). A block can be decomposed in sub-blocks or in a set of communicating processes (see Fig.3b). A process is implemented as an extended state machine, where the communication semantics is defined by means of a single buffer of infinite length for each state machine. SDL state machines are a mixture of state diagrams and flow charts (see Fig.3c); in fact, they present states, transitions and selections (equivalent to the "if" statements of high-level languages). In SDL state diagrams, reading of inputs and writing of outputs, as well as the execution of assignments and procedures can be associated with each transition. Reading should always precede writing, since inputs usually represent the condition for transition. If no input reading is defined, the change of state is always performed. For example, the exiting from the initial state is usually performed without reading any input (see Fig.3c).

In SDL, timing constraints are modeled through *timers*. A timer can be considered as a separate process which is able to send messages. A process can have a set of timers, which can be set, read or reset. For example, an SDL state machine that must satisfy a timeout must set a timer and then wait for the message signalling its occurrence. To this end, the state machine must be able to receive input messages from the timer in all its states, and a transition from a state to the next cannot be interrupted. As a result, definition of deadlines may result in somewhat complicated expressions. SDL permits the dynamic generation of processes; this is denoted with a dashed line as in Fig.3b. For the above reasons, the behavior of an SDL specification can be non completely deterministic. On the other hand, the validation is performed by simulation and, thus, the analysts achieve the final version by refinement.

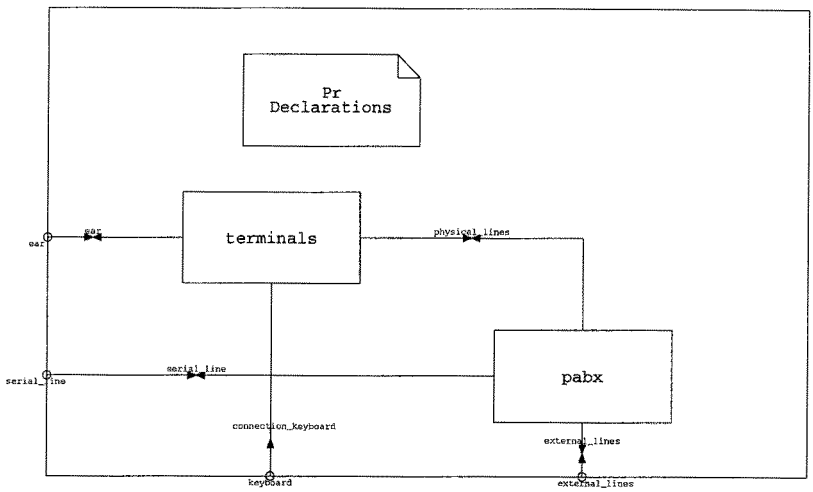

Figure 3a. Example of system specification in SDL 1988: system decomposed in two blocks *terminal* and *pabx*.

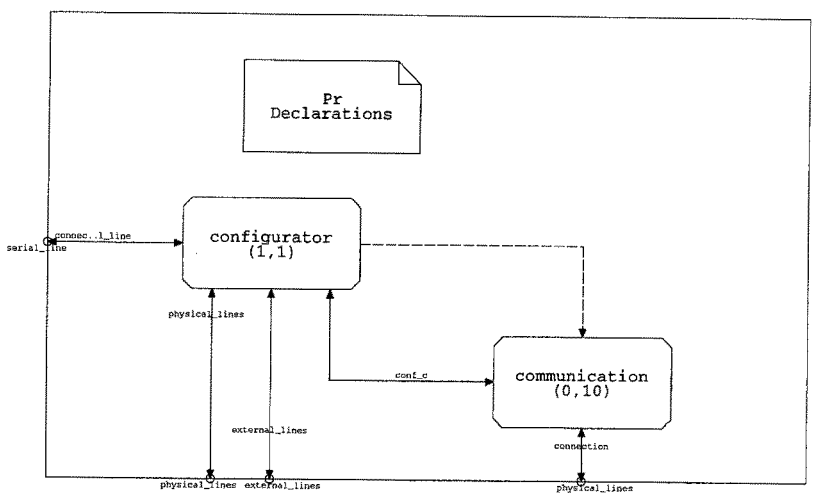

Figure 3b. Specification in SDL 1988: the block *pabx* is decomposed in the processes *configurator* and *communication*.

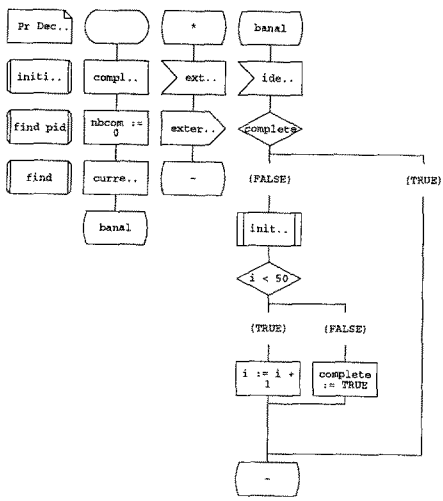

Figure 3c. Specification in SDL 1988: a part of the implementation of process *configurator* as an SDL state machine where: *banal* is a state, the empty rounded box is the initial state of the machine, *ext..* and *ide..* are inputs, *ester.*. is an output, simple squared boxes contain assignments, etc..

More recently, an object-oriented extension of SDL (i.e., SDL 1992) has been presented (Braek and Haugen, 1993). This integrates the classical notation of SDL with the power of the object-oriented paradigm (e.g., Object-oriented SDL, OSDL). The major advantage is the presence of the mechanisms of inheritance, polymorphism and instantiation which have been defined for both the block and state machine levels. For example, new blocks and processes can be defined and used in more than one place (i.e., instantiation).

There are many SDL tools — e.g., GEODE (GEODE, 1992) — which cover all features defined in the so-called SDL 88 (version of 1988). Moreover, for supporting the configuration management, versioning, and report generator, other instruments are needed. In order to manage the problems of real-time in telecommunications supporting functional, behavioral and structural aspects, several extensions of SDL, such as (Encontre, et. al, 1990), have been presented.

3.1.3. Esterel

Esterel is an operational programming language introduced in 1985, which supports a set of elementary instructions such as: *loop* (indefinite), *if-then-else*, etc. Among these, there are special instructions for defining expressions of timing requirements (Berry and Cosserat, 1985). A program specifies the deadline for procedure execution and let's suppose that the requirements are met at run-time. Consider for instance the following piece of code:

```
var A, B: int in;
    loop
        do
            A := Fun(B)*5.6;
            .....;
            .....;
        uptonext 10 seconds end;
    end;
end;
```

where: A, B are variables, and *seconds* is a signal. For this program, the instructions inside the **do-uptonext** body are executed for the next 10 *seconds*. By using the elementary instructions, more complex instructions can be defined.

Rather than requiring that all timing behavior be known at compile time, Esterel allows the programmer to specify not only the timing requirements, but also allows the definition of the exception handlers which will be executed if the timing requirements are not met (recovering from time-failure). This approach to validation is very similar to that adopted by SDL. The execution is obtained by translating the Esterel specifications in communicating finite state machines. The number of processes is finite and their communications are through broadcasting. The execution model is synchronous and communications are considered instantaneous. These assumptions imply that the communications can be simply described by a discrete history of events where several events

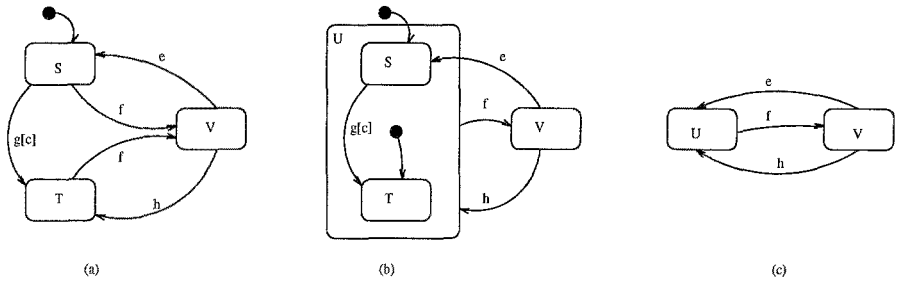

Figure 4. Statecharts notation and the operation of state clustering (i.e., the XOR) (a) Traditional state diagram, (b) Statecharts notations at detailed level, (c) Statecharts notations at abstract level.

can formally occur at the same time instant. As a result, if a message is not received at the same instant in which it is made available, it is lost. Esterel makes no provision for exhaustive static analysis before compilation; therefore, it does not ensure predictability, in the strong meaning of completing without exception. This is essentially due to the adoption of a strongly synchronous model with instantaneous communications.

3.1.4. Statecharts

Statecharts have been firstly introduced in 1987 as a visual notation for representing complex state machines, in a more synthetic manner with respect to the usually adopted notations based on state diagrams (Harel, 1987), (Harel, 1988). With this notation, complex state machines are represented as combinations of simpler machines, through the XOR and AND mechanisms as shown in Fig.4 and Fig.5, respectively. In this way, the explosion of the number of states of conventional state diagrams is strongly reduced. On the other hand, this notation may be less intuitable than conventional state diagrams.

Associated with the notation, an operational semantics has been presented, which describes how single machines are executed in order to model the equivalent complete state machine (Harel, 1987). Following this semantics, the single state machines are considered as concurrent and communicating through broadcasting (similar to Esterel). A state machine can observe both the current status of other state machines and the history of their behavior by using special functions provided by this language. The operational semantics is based on a set of micro-steps in which the execution of a single state machine is decomposed. In some cases, Statecharts can lead to define non-deterministic paths of execution.

In Statecharts, the notion of time is managed through the special function *timeout(E,N)* which becomes true when N time units are passed, after the last occurrence of event E (STATEMATE, 1987). Statecharts have been defined for modeling only system behavior, they can be profitably used as a specification language only if they are integrated in a CASE tool where the structural and functional aspects are addressed by

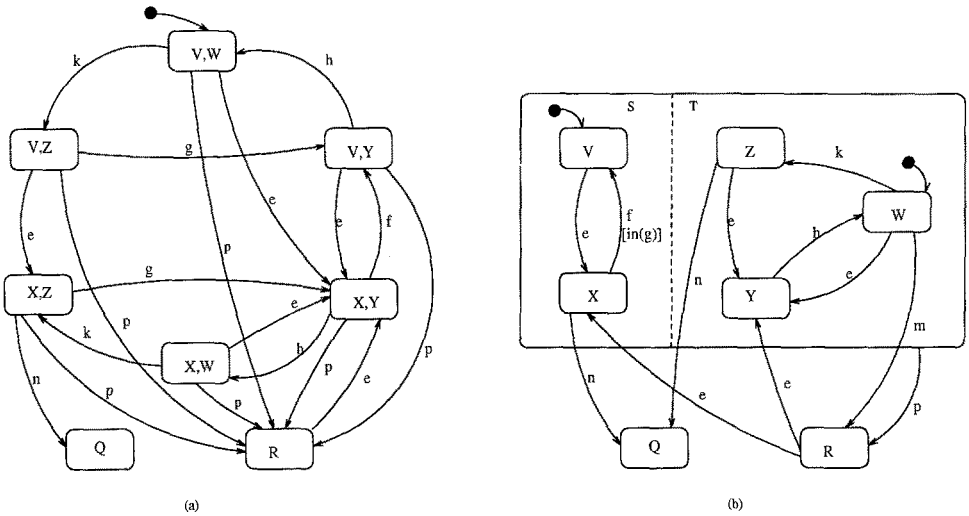

Figure 5. The operation of state machine composition (i.e., the AND operation) of Statecharts (a) Traditional state diagram, (b) Statecharts notations where S and T are orthogonal components of the complete state machine.

means of other notations. This has been done with STATEMATE (STATEMATE, 1987), (Harel, et. al, 1990). This tool makes an explicit distinction among structural, functional and behavioral aspects. These three aspects are described via three different representations, that is, activity-chart, statechart, and module-chart — modeling functional, behavioral and structural aspects, respectively. The activity-chart is a sort of RT-DFT (see Section 3.3.1), while the module-chart is a visual notation for structural decomposition. STATEMATE controls that consistency and completeness are maintained through the three different notations. STATEMATE has the capability of verifying the correctness of a Statecharts by means of exhaustive and sub-exhaustive execution tests. The verification of reachability, the presence of non-deterministic conditions and deadlocks, and the use of transitions, are identified through these tests. Simulation gives the system a great confidence in producing specifications.

More recently, several extensions of the Statecharts model have been proposed for improving its capabilities in modeling timing constraints and functional aspects — e.g., Modecharts (Jahanian and Stuart, 1988), Objectcharts (Coleman, Hayes and Bear, 1992), and ROOMcharts (Selic, 1993). In Objectcharts the model of Statecharts has been ported in an object-oriented environment. The concepts of temporal constraints on state transitions and those of auxiliary variables for state machines have also been added (see Fig.6). Object orientation has solved the problems related to the different views by integrating them in the concepts of classes (objects). Moreover, other classical object-oriented concepts such as inheritance, and instantiation have also been added. ROOMcharts is also supported by an object-oriented methodology (Selic, et. al, 1992) and a CASE tool named ObjecTime (NorthernTelecom, 1993). The Modecharts model is also discussed in Section 4.2.1.

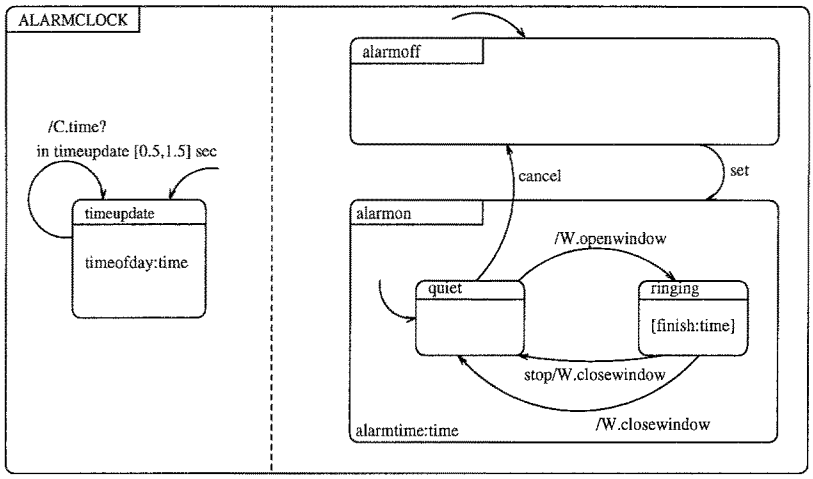

Figure 6. Example of an alarm clock in the Objectcharts notation, where *alarmtime* of type *time* in the state machine *alarmon* is an auxiliary variable.

3.1.5. Other Operational Approaches Based on State Machines

In this section, further interesting approaches based on state machines will be discussed for completeness. The classical model of extended state machine has been modified to offer the capability for defining timing constraints — e.g., RTRL (Real-Time Requirements Language) (Taylor, 1980), ESM (Extended State Machine, which will be discussed in Section 5.1) (Ostroff and Wonham, 1987), and CRSM (Communicating Real-time State Machines) (Shaw, 1992).

RTRL was firstly developed by the GTE laboratories, for their internal use (Taylor, 1980). The notion of time is modeled through the concept of timer as SDL (on the other hand, minimum and maximum time constraints on the occurrence of events can be defined in terms of timers (Dasarathy, 1985)). To avoid the problems of several other languages which assume the execution time to be instantaneous, in RTRL, time durations (such as the execution of sending/receiving a signal) are modeled by means of dedicated constructs which consider the execution time.

CRSM is an extension of the classical CSP model, which adds timing constraints on the conditions for the execution of transitions (Shaw, 1992). In particular, the minimum and the maximum time in which the transitions can be enabled is defined. Time is considered as continuous; therefore, it is represented as a real value (in floating point). The value of the current real time is available as the common variable *rt*. In CRSM, the system is described by means of a set of communicating state machines in which communications are strictly synchronous, even if asynchronous communications can be defined by resorting to infinite buffers on the inputs. The system validation is performed via simulation, and by analyzing the history traces.

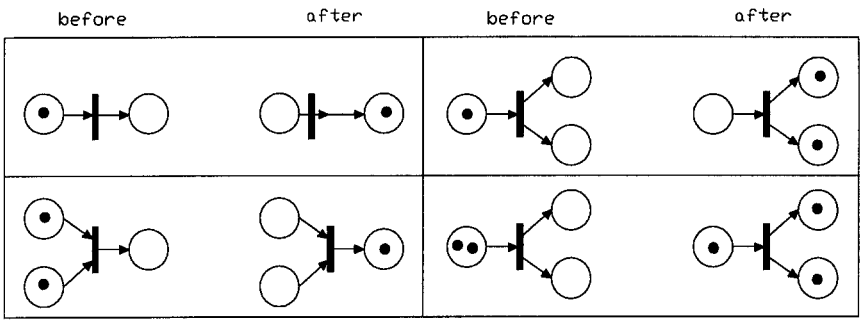

Figure 7. Example of the execution of simple Petri Nets.

3.2. Operational Approaches Based on Petri Nets

Petri Nets (PNs) were introduced by Petri as early as 1962 (Petri, 1962). They are more expressive than state machines, which can be proved to be a Petri nets subclass (Murata, 1989). However, given their lower intuitability, PNs have hardly found their way into programming languages.

PNs are an operational formalism, especially suitable for modeling synchronizations among asynchronous processes (Merlin and Faber, 1976), (Reisig, 1985), (Peterson, 1981). A Petri net is a graph comprising a finite number of *places* (circles) and *transitions* (segments) (see Fig.7), connected by oriented arcs. A set of *tokens* can be associated with each place. The state of a PN corresponds to the distribution of tokens on the places (i.e., the *marking* of the net). The operational semantics is that the presence of places with at least a token connected with arcs to a transition makes it "firable" — i.e., executable. The execution of a transition leads to the generation of a token in each place connected by an arc going out of the transition itself. In Fig.7, several examples of executions are reported. When a transition has more outgoing than ingoing arcs, it is a producer of tokens, and when the number of outgoing arcs is lower than that of ingoing arcs, the transition is a consumer of tokens. A review of fundamentals about Petri nets can be found in (Murata, 1989).

A PN is *safe* if the number of tokens is limited in time; in this case, it can be easily transformed in a finite state machine which is called *Token Machine*. On the contrary, if the PN is *unsafe* it corresponds to a state machine having an infinite number of states, defining in this way a divergent behavior. In this case, the verifiability of a PN is impossible. If the number of tokens of a PN is constant for each state, the PN is called conservative. On the contrary, PNs are called *inconsistent* when the behavior (i) terminates for token consuming, or (ii) diverges for an uncontrolled production of tokens.

The verification of a PN is based on the analysis of the Token Machine. All the classical verifications, which can be executed on finite state machines to verify reachability, deadlock free, etc. are performed on the Token Machine. If in a PN there is more than one transition enabled by the same *marking*, the execution is non-deterministic (i.e., the PN has more than one possible next state). PNs in which this condition never occurs

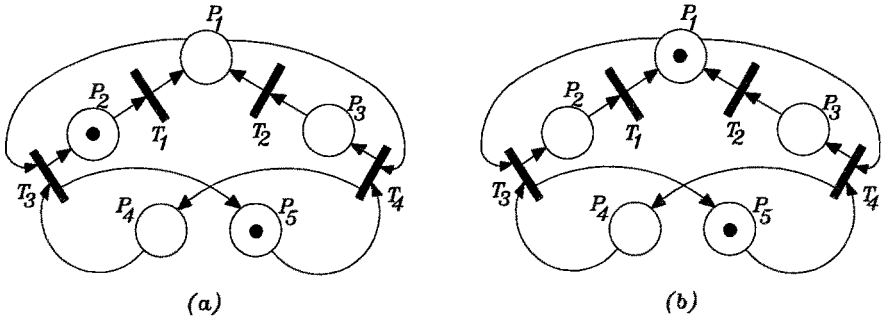

Figure 8. Example of Petri Nets, where T_i are transitions, P_i places, (a) and (b) are two consequent states of the net.

behave in a determinism manner and are called *decision free* nets. The verification of a PN is computationally possible only for *consistent deterministic* nets, while for the non-deterministic nets the problem is NP complete (Willson and Krogh, 1990). For conservative nets, the absence of deadlocks can be demonstrated by using algorithms with exponential complexity. On the other hand, these algorithms cannot be extended to all the extensions of the classical Petri net model.

It should be noted that, a state diagram can be regarded as a particular Petri net where the transitions have only one entering and one outgoing arc. On the other hand, the state diagrams are more intuitive with respect to the visual representation of Petri nets. The classical notation for Petri nets is unsuitable for representing the functional and structural aspects of the system under specification; there is no explicit support for specifying timing constraints, and the tokens are only untyped control messages. For these reasons, many extensions have been proposed for transforming PNs into suitable models for the specification of real-time systems. The following subsections are dedicated to the discussion of several of them.

3.2.1. TPN

The TPN (Time Petri Nets) model has been introduced in 1976. It is an extension of classical Petri nets for treating the timing constraints. These are expressed explicitly for each transition, by means of a minimum and a maximum time ($Tmin$ and $Tmax$, respectively) (Merlin and Faber, 1976). $Tmin$ is the minimum time for which the transition must stay enabled in order to be firable. $Tmax$ is the maximum time for which the transition can stay enabled without firing. If $Tmin = 0$ and $Tmax = \infty$ the TPN corresponds to the classical PN model. The adoption of this time modeling allows the definition of timeouts, deadlines, etc. A TPN, defined on a structurally safe PN, can be verified for controlling the absence of deadlocks and other properties (Berthomieu and Diaz, 1991), (Leveson and Stolzy, 1987). The Token Machine resulting from a TPN is different from that of the corresponding PN (i.e., the same TPN without timing constraints) and may

present unreachable states, thus requiring a more accurate analysis. Moreover, in certain conditions, the generation of the Token Machine and the reachability analysis can be impossible.

3.2.2. SPN

SPNs (Stochastic Petri Nets) are an extension of the classical PN for describing the timing constraints, introduced in 1983 (Marsan, Balbo and Conte, 1983), (Molloy, 1985). In this model, a random variable is assigned to each transition T_i representing the firing delay. In this way, in the presence of several firing conditions in the nets, these are ordered by means of their respective firing delays. SPNs with geometrical or exponential distributed delays are isomorphic to homogeneous Markov chains; therefore, they are appropriate for modeling non-deterministic processes.

PROT nets are an extension of SPN and allow the discovering of critical conditions (Bruno and Marchetto, 1986). These can be easily translated in ADA language, where the interactions between a process and a transition take place through two rendez-vous of ADA. PROT nets are an efficient system for producing the ADA structure of a system by specifying system behavior. For these nets, a simulator is also available to validate the specification. This model is also supported by a CASE tool, named ARTIFEX (ARTIFEX, 1993), which includes the aspects of object-oriented paradigm and methodology called PROTOB (Baldassari and Bruno, 1991).

3.2.3. Other Operational Approaches Based on Petri Nets

For completeness, other approaches based on Petri nets are briefly discussed in this sub-section, in the order of their appearance — i.e., Timed-PN (Timed Petri Nets) (Ramachandani, 1974), CPN (Colored Petri Nets) (Jensen, 1981), (Jensen, 1987), HMS (Hierarchical Multi-State machines) (Gabrielian and Franklin, 1991), ER, TER and TB nets (Ghezzi, et. al, 1991), and CmPN (Communicating Petri Nets) (Bucci, Mattoline, and Vicario, 1993), etc.

In Timed-PNs, a duration time is associated with each transition for modeling the execution time of the transition itself (Ramachandani, 1974), (Ramamoorthy and Ho, 1980). The semantics of classical PNs is modified by assuming that a transition must fire as soon as it is enabled. These nets are mainly suited for performance evaluation.

The HMS model is based on state machines, but presents many characteristics which make this formalism more similar to Petri nets than to state machines (Gabrielian and Franklin, 1991). In fact, as in Petri nets, in a HMS specification several active states (i.e., states with a sort of "token") can exist at the same time. In HMS the system under specification is modeled as a hierarchy of specifications starting from the more abstract to the more detailed one. This layering allows the reduction of diagram complexity, similar to the clustering mechanism of Statecharts formalism (see Section 3.1.4). Moreover, the specification refinement is supported by a process of verification which validates if the conditions (expressed by means of axioms) for any given abstract level are satisfied by

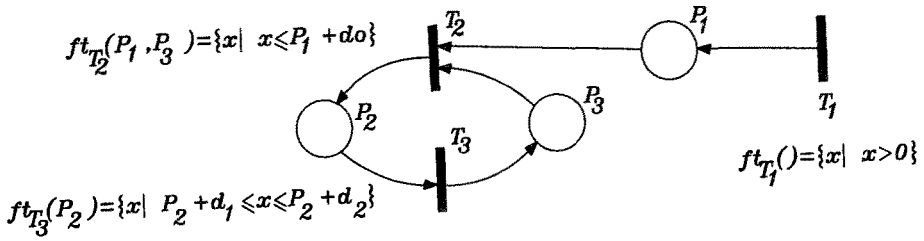

Figure 9. Example of a TB net.

the specification for the lover level (i.e., more detailed). Timing constraints are defined in Temporal Interval Logic (TIL). Timing constraints can be used for defining conditions on state transitions. The complete model is non-deterministic, but executable, and thus it is suitable for simulation.

In (Ghezzi, et. al, 1991), a collection of extended Petri nets has been presented in order of complexity: ER, TER and TB nets. In the TB model, the time in which a token has been produced (i.e., *timestamp*), is associated with the token itself. In TB nets, tokens are "functions" which associate values with variables. In addition, a Boolean condition based on the presence of tokens with their *timestamps* on the connected places is associated with each transition. The TB model is to be a generalization of the TPN model proposed in (Merlin and Faber, 1976). The transformational aspects (i.e., *functional aspect*) of the system under specification can also be described by the TB model. Fig.9 reports an example presented in (Ghezzi, et. al, 1991). It represents a net that models a data acquisition system which periodically samples data from the environment. Sampled data are modeled by the tokens fired by transition T_1. A controller (represented by a token in place P_3) takes those data (i.e., transition T_2 fires) and then elaborates them, i.e., transition T_3 fires. The sampled data are valid only at most for d_o time units and the elaboration takes a minimum of d_1 and a maximum of d_2 time units.

CmPNs are an object-oriented extension of PN. These are suitable for modeling structural, behavioral aspects of the system under specification (Bucci Mattoline and Vicario, 1993). By means of this model the system is regarded as a set of asynchronously communicating subsystems (which can be placed in a single-processor or distributed environment). In the CmPN model, a *priority* and an *action* are associated with each transition. The scheduler takes into account the priority during its work, and the action corresponds to the execution of a procedure when the transition is fired. For a specification given in terms of CmPNs, the absence of deadlocks can be verified. Moreover, a CmPN specification can be directly translated into C++ code for a heterogeneous environment based on MS-DOS and UNIX machines.

3.3. Operational Approaches Based on Other Notations

Most of the operational models have their own visual notations and this increases specification understandability. Starting with the late 1970s, a number of different notations

were introduced as a substantial component of various methodologies for system analysis and design. Most of them, including Structured Analysis (Ross and Schoman, 1977), (Yourdon and Constantine, 1979), Data Flow Analysis (DeMarco, 1979), JSP (Jackson Structured Programming) (Jackson, linebrak 1975), and JSD (Jackson System Development) (Jackson, 1983), have found large acceptance by industry. In addition, because of their expressiveness, these notations have been incorporated in many CASE environments. As a result, the boundaries between the use of these notations and of operational visual specification languages are crumbling. In fact, tools supporting methodologies of the above mentioned category become increasingly similar to visual programming environments, thus allowing the direct manipulation of graphical elements in order to visually describe program aspects like structure, flow of data and the like. Following this trend, conventional approaches like Structured Analysis, have be considered operational, in spite of their informal nature.

The use of visual techniques reduces the effort of user-machine communication. In this way, the ability of the user to describe the system is greatly enhanced, further reducing the intermediation with the system. This aspect has been taken into consideration by the designer of tools like those described in (Jacob, 1985) and (Harel, 1988).

In recent years, the Object-Oriented Paradigm (OOP) has gained large acceptance in software analysis and design (Coad and Yourdon, 1991), (Booch, 1991), (Northern-Telecom, 1993), (Coleman, Hayes and Bear, 1992). This is largely due to the fact that working with classes and objects is an easy and natural way for partitioning large problems. The mechanisms for decomposing a system into objects, makes OOP the natural methods for separating the different activities that can be carried out in parallel. By supporting strong modularity, code reusability, and extendibility, OOP is having quite an impact on design, implementation and maintenance of complex systems, and many formal languages have been extended with object-oriented capabilities, leading to languages like Z++, VDM++, TRIO+, etc. Several object-oriented methodologies for real-time systems have also been introduced — e.g., HOOD (HOOD, 1988), Wirsf-Brock et al. (Wirsf-Brock, Wilkerson and Winer, 1990), Booch (Booch, 1991), OMT (Rumbaugh, et. al, 1991), Coad and Yourdon (Coad and Yourdon, 1991), and these have been adopted as a basis for a number of CASE tools.

3.3.1. Structured Analysis

Structured Analysis is a generic term which denotes a number of analysis and design methodologies, approaching system specification in a *structured* manner.

Historically, SADT (Structured Analysis and Design Technique, introduced in 1977) (Ross and Schoman, 1977) has been the first methodology based on a structured approach and, more specifically, on the concept of functional decomposition. SADT has its own diagrammatic notation, with well-defined syntax, semantics. The notation handles the data and control flows. SADT builds on the concept of *models*. A model corresponds to a hierarchy of diagrams describing the system from a particular point of view (for instance, from the operator's point of view). Models can share common details. A sub-model which is shared by other models or whose implementation can be changed is

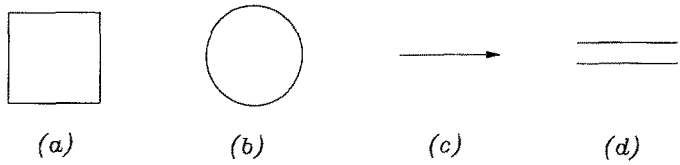

Figure 10. Symbols of DFD: (a) external agent, (b) function or process for data transform, (c) data flow, (d) permanent data repositor.

called a *mechanism.* A mechanism can remain undeveloped until the later stages, thus delaying detailed decisions. SADT is a registered trademark of SofTech and is available commercially.

Structured analysis as publicized by Yourdon and Constantine (Yourdon and Constantine, 1979) and De Marco (DeMarco, 1979) in 1979, is perhaps one of the best known methodologies (Svoboda, 1990). This is essentially a functional decomposition technique, used to identify a number of transforms and the data flowing among them (DeMarco, 1979). This is done by the successive application of the engineering definition of a black box that transforms an input data stream in an output data stream. Data-Flow Diagrams (DFD) and data dictionaries are the most important tools used in carrying out the analysis. Transforms are called processes and represented as bubbles in data flow diagrams, while flows of data are represented as oriented arcs among bubbles. The data dictionary keeps track of any data flowing through the network of processes. The graphic symbols used in Structured Analysis/Data-Flow Analysis are shown in Fig.10.

Data-flow analysis tends to produce a network of (sub)programs corresponding to each transform, but allows the identification of the central transform, that is, the one that accepts the most abstract input stream and produces the most abstract output stream. The central transform corresponds to the most abstract view of the program functionality. By picking up the data-flow diagram from the central transform, a tree representing the hierarchical structure of the program is obtained. Starting from the so-called context diagram, which contains only a single bubble representing the system in its entirety, successive refinements are applied (DeMarco, 1979). This leads to a hierarchical decomposition, where each process is decomposed in a number of lower-level, more detailed data transforms. Of course, consistency must be kept among levels. In particular, input/output data, flowing in/from a given process, must be preserved when the process is decomposed in a number of lower-level processes.

Structured Analysis can be applied in a purely manual fashion or it can be automated to varying degrees (Birrel and Ould, 1985). Almost any CASE tool produced in the last decade includes Structured Analysis — e.g., (Teamwork, 1992), (StP, 1991). In addition, DFDs admit also an operational interpretation.

Structured analysis is still one of the most used techniques for dealing with transformational applications, as found in data-processing environments, where procedural (i.e., static) aspects are relevant. Its strength is also its weakness: it is informal and very intuitive, and thus can be used also by people not keen in mathematics; however, this denies rigorous specification validation.

3.3.2. *Real-Time Extensions of Structured Analysis*

Several extensions to structured analysis have been proposed in the 1980s, in order to take into account also system dynamics and use it for the specification of real-time systems.

DARTS (Design Approach for Real-time Systems) (Gomaa, 1984), (Gomaa, 1986), and its recent object-oriented extension (Gomaa, 1992), is an example of design techniques which closely follow the Structured Analysis/Data-Flow analysis approach to identify the processes to be implemented in a real-time system, as well as their synchronizations.

Ward and Mellor (Ward and Mellor, 1985), (Ward, 1986), extended data flow diagrams by adding edges representing control. They also used state machines for representing behavior. Many commercial CASE tools, including Teamwork (Teamwork, 1992), StP (Software through Pictures) (StP, 1991) and Excelerator (Excelerator, 1986) have followed this lead.

Hatley and Pirbhai (Hatley and Pirbhai, 1987) performed a data-flow analysis and then proceeded with a control-flow analysis. This leads to augmenting the data flow-diagram with the so-called control bars, which are introduced to represent event occurrences. Additional specification tools (i.e., CSPECS (Hatley and Pirbhai, 1987)) are used in order to express how and when a transformation occurs. As a result, the designer must employ different tools and languages, depending upon the stage of the analysis. In Fig.11a, the top level of a cruise control system according to the Hatley and Pirbay notation, is reported. In the diagram, both data and control flows are shown, where square boxes represent terminal objects. In Fig.11b, the DFD diagram at level 0 shows the decomposition of the system in "processes" (according to Hatley and Pirbay, but called data transformations in the more general meaning), data and control flows (i.e., continuous and dashed lines), data stores (e.g., *mile count*) and a control bar (i.e., *CNT_1*). The control bar encapsulates the system behaviour and process activation in terms of a Mealy state machine (see Fig.11c).

Structured analysis for real-time systems is still based on the notion of the flow of data between successive transforms, and provides little support to identify the concurrent processes (in this paper, the word "process" was used to denote a separate sequential activity, implemented as an independent thread of execution, and not simply a data transform). that must be implemented in a given application. Depending upon the detail of the analysis, there is something arbitrary in identifying the system processes. This may result in the implementation of unnecessary processes and the possibility that a given process needs concurrency internally.

3.3.3. *JSD and Entity Life*

Jackson System Development (JSD) (Jackson, 1983), (Cameron, 1986) was introduced in 1983, and has become since then a well-known method for software development. It encompasses most of the software life-cycle and, being based on the concept of communicating sequential processes, it can be used for the design of real-time and concurrent software.

JSD is an entity-life modeling approach (Sanden, 1989c) to software analysis and design. In fact, the first phase of the method, the model phase, is devoted to the examination

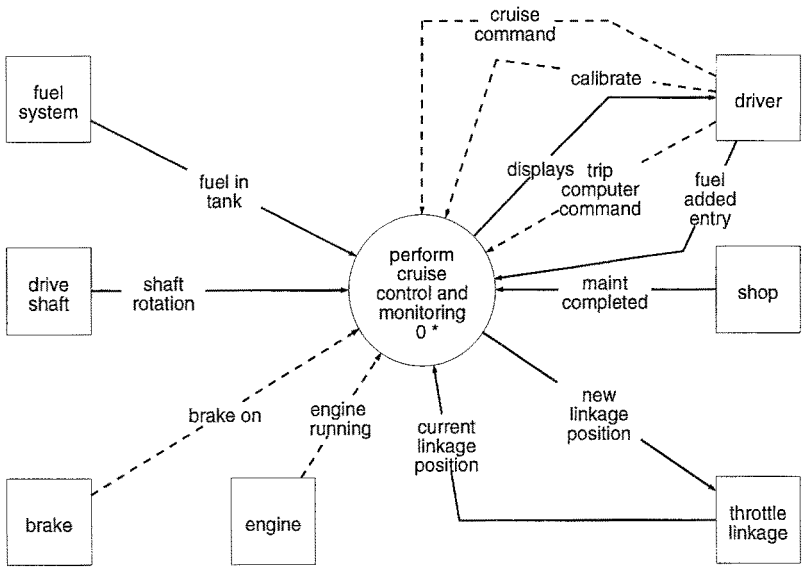

Figure 11a. Example of extended DFD in the Hatley and Pirbay notation: Context Diagram.

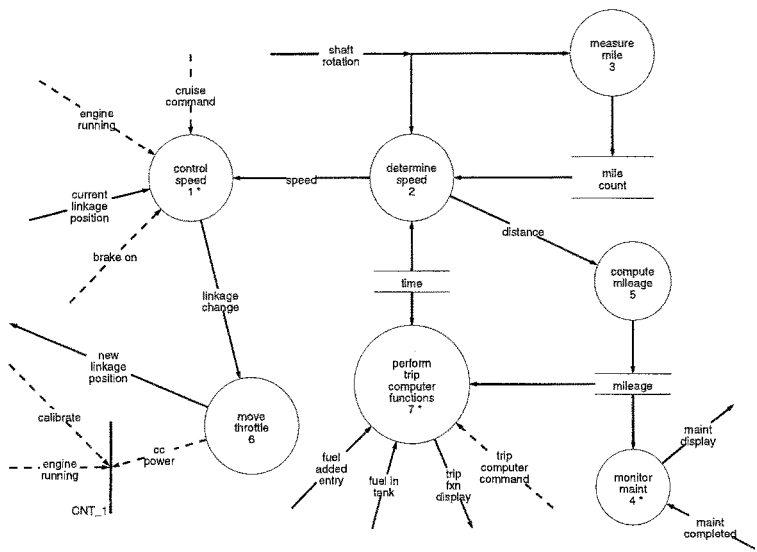

Figure 11b. Example of extended DFD in the Hatley and Pirbay notation: level 0.

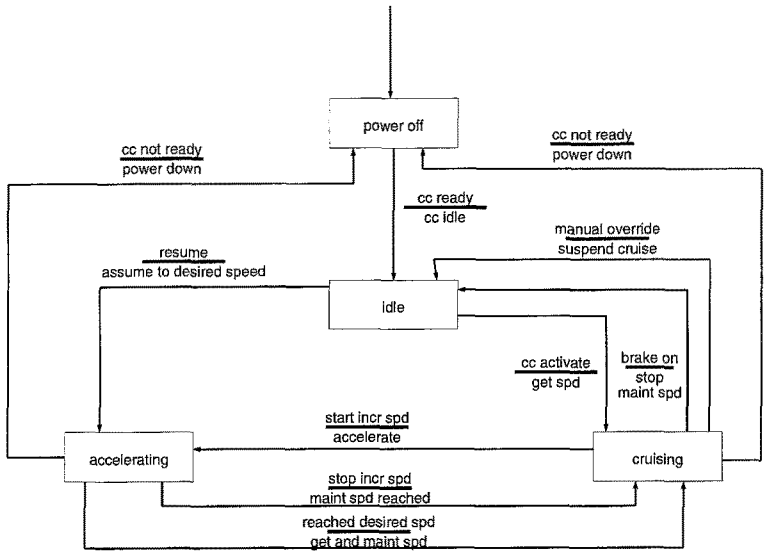

Figure 11c. Example of extended DFD in the Hatley and Pirbay notation: behavior of *control speed.*

of the real-world environment, in order to identify and model the entities belonging to it. Entities that have a strong time dimension, that is, when they make transitions between different states over time, are modeled as software processes. Modeling is done trough an explicit diagrammatic notation in which three basic structuring concepts are used, namely: sequence, iteration and selection. The resulting entity structure describes all the possible life histories of the entity itself. Since JSD makes reference to languages that do not support concurrent programming, processes are implemented as coroutines, managed by a tailor-made scheduler.

Elaborating upon JSD, Sanden has proposed the so-called generic entity-life approach to concurrent software design (Sanden, 1989c), (Sanden, 1989b), (Sanden, 1989a). In this approach, the first step is the identification of each independent and asynchronous thread of events in the problem domain; for any thread of events, a software process is implemented in the system. The generic entity-life approach avoids certain intricacies of JSD, as well as the implementation of unnecessary processes. Furthermore, Sanden uses the ADA language to work out his examples. The task construct of that language avoids another problem of JSD, that is, the implementation of processes as coroutines and the consequent need of a tailor-made scheduler.

Entity-life approaches are a step towards clear specifications of concurrent systems. They take into account functional, structural and behavioral aspects. In the examples presented in (Sanden, 1989c) and (Sanden, 1989b), system behavior is dealt at the level of interactions among tasks. The specification of internal details is left to other techniques, such as state machines.

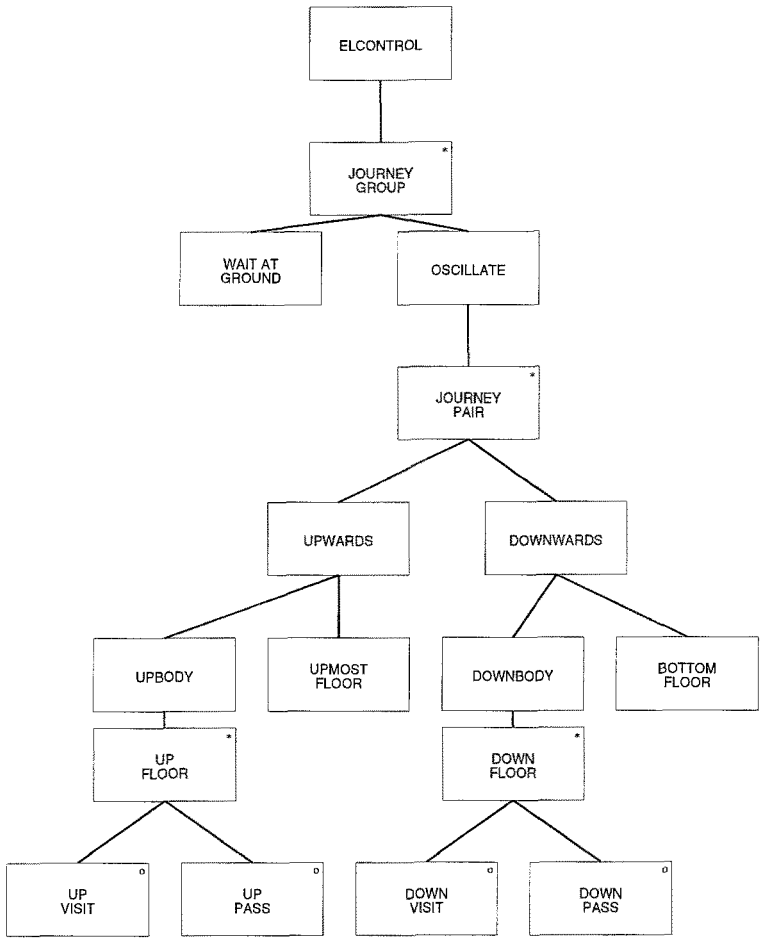

Figure 12. Structure of control software of an elevator in JSD. Operations, represented by square boxes are described by decomposition; at the same level they are executed from left to right; iterative executions are marked by "*"; and selections with an "o".

3.3.4. *Object-Oriented Notations*

In recent years, the Object-Oriented Paradigm (OOP) has gained a large acceptance in the software community. Object orientation has also been used for the development of real-time systems, due to the fact that the object-oriented model can be considered intrinsically concurrent. In fact, software objects can be regarded as independent threads of execution which communicate by means of message passing (Cox, 1984), (Diederich and Milton, 1989), (Bihari and Gopinath, 1992). More refined models divide system objects into active and passive, and/or server and client objects (Agha, 1986), (Ellis and Gibbs, 1989), (Booch, 1991). For these reasons, both object-oriented methodologies and languages (e.g., (Ishikawa, Tokuda and Mercer, 1992)), have been defined for modeling real-time systems. Since, this section is devoted to the operational approaches, only methodologies are discussed. Pure languages are not mentioned, even those that are focused on programming distributed and/or real-time systems.

In the early 1990s, many efforts have been made to reuse the good things of the old conventional methodologies, such as DFD, Entity-Relationships diagrams (ER) (Chen, 1976), etc., by reinterpreting them in the context of an object-oriented methodology — e.g., Coad and Yourdon (Coad and Yourdon, 1991), Rumbaugh at al. (OMT) (Rumbaugh, et. al, 1991), and Martin and Odell (Martin and Odell, 1991). Of course, the resulting techniques are influenced by the functional view. More recently, some "pure" object-oriented methodologies have been proposed — e.g., Booch (Booch, 1991), and Wirsf-Brock et al. (Wirsf-Brock, Wilkerson and Winer, 1990). Pure object-oriented methodologies focus only on the definition of objects and relationships among them (Monarchi and Puhr, 1992).

In many of the above-mentioned approaches, the system is decomposed into objects for representing the structural aspects of the system under specification. Object relationships are defined through extended Entity Relationship diagrams (Coad and Yourdon, 1991), (Rumbaugh, et. al, 1991) or by using the so-called Object Diagrams (see Fig.13) (Booch, 1991), (Rumbaugh, et. al, 1991), (HOOD, 1988), (Wirsf-Brock, Wilkerson and Winer, 1990). To support all the features of the OOP, such as inheritance, polymorphism, aggregation, association, etc., special symbols for Entity Relationship diagrams or special diagrams, such as Class Hierarchy, have been defined (see Fig.14). In most of the proposed methodologies, system behavior is encapsulated in the implementation of objects (more specifically in the implementation of class methods). The object behavior is usually described by means of extended state diagrams or state transition matrices. Shlaer and Mellor (Shlaer and Mellor, 1988), (Shlaer and Mellor, 1991) and Booch (Booch, 1991), use a Mealy model; Rumbaugh (OMT) (Rumbaugh, et. al, 1991) uses a notation strongly similar to Statecharts — i.e., the ROOMcharts (see Section 3.1.4); Coad and Yourdon (Coad and Yourdon, 1991) use a state event table.

It should be noted that, the description of the class interface in terms of methods is not able to represent all the relationships that objects may have with respect to other objects, especially in a concurrent environment (Bucci, et. al, 1993), (Coleman, Hayes and Bear, 1992), since this does not represent the services that a class of objects *requires* from other objects (Wirsf-Brock and Wilkerson, 1989), (Walker, 1992). In fact, these requests are encapsulated into the methods body and, thus, they are hidden to the outer objects.

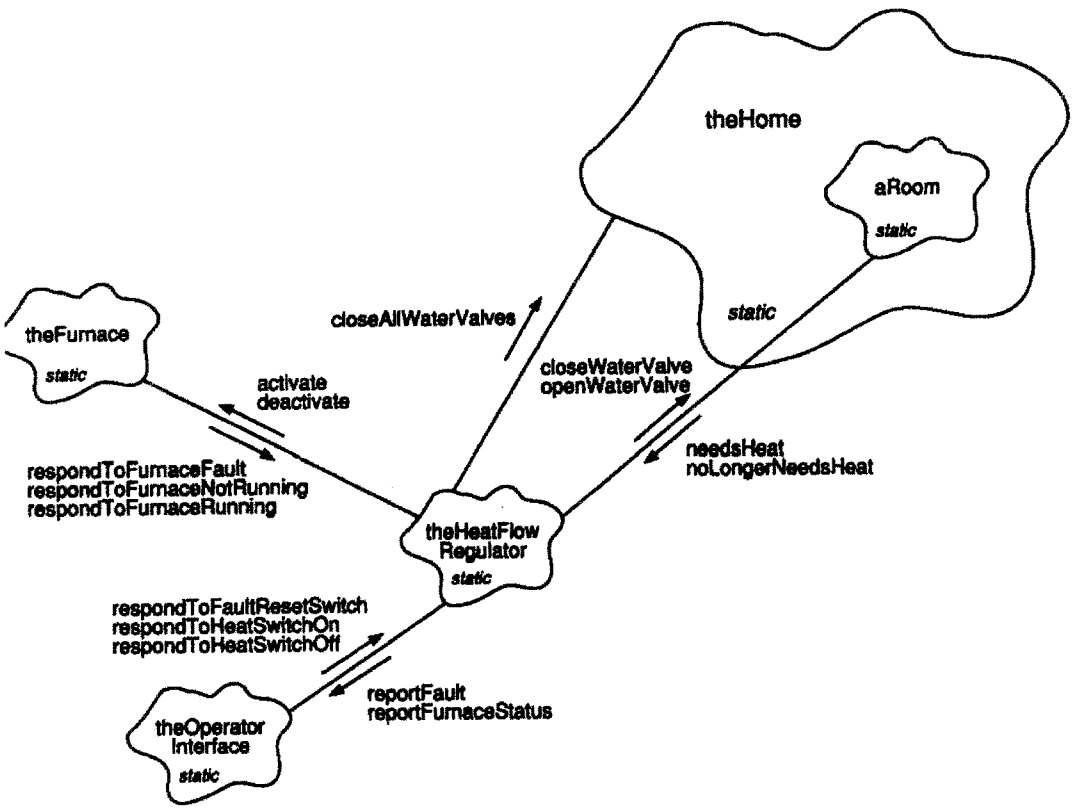

Figure 13. Example of a home heating system, Object Diagram in Booch's notation.

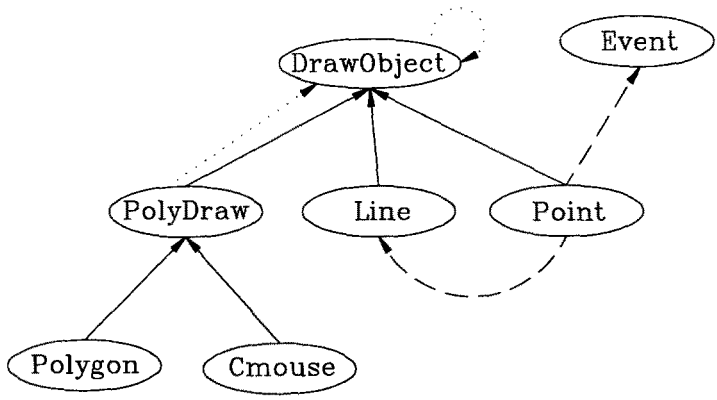

Figure 14. Example of Class Diagram (also called class tree — DrawObject is the root), where continuous lines define relationships of specialization between classes (i.e., inheritance) (*is_a*), dashed lines those of aggregation (*is_part_of*), and dotted lines those of association (*is_referred_by*).

Furthermore, though many of these methodologies are especially defined for the analysis and design of reactive systems, some of them are not completely satisfactory for specifying real-time systems. Usually, these notations only provide support for defining timing constraints of the system under analysis, but unfortunately they are not strongly supported by techniques for verifying the consistency and completeness of time relationships. This derives from the fact that these methods are not enough formal for supporting a formal semantics and for defining an executable model of the system.

In spite of the previous considerations, many CASE tools have been built on the basis of the above mentioned methodologies. In these CASE tools, model limitations have been partially circumvented through specific notations which add formalism and executability — e.g., object-oriented and state machines (e.g., Shlaer and Mellor (Shlaer and Mellor, 1988), Booch (Booch, 1991)), object-oriented).

4. Descriptive Approaches

The use of descriptive languages for program specification has been proposed by several researchers during the 1970s — e.g., (Guttag, 1977), (Liskov, et. al, 1977). Descriptive approaches are based on mathematical notations (axioms, clauses, set theory, etc.) and produce precise, rigorous specifications, giving an abstract view of the system state space. The system is described by specifying its global properties, forcing the analyst to specify *what* must be done by the system rather than *how* it must be done. Descriptive specifications can usually be automatically processed for verifying their completeness and consistency. Moreover, a specification can also be validated by proving that high-level properties are verified by the specification itself. This is performed by means of theorem provers or Prolog engines. Since most of these are not enough efficient and predictable (from the performance point of view), descriptive approaches are not

considered adequate for producing executable real-time specifications. Only in the late 1980s, some descriptive languages have been enriched with primitives for dealing with time, making them suitable for specifying real-time systems.

In the following, we have tried to classify descriptive approaches on the basis of their main nature, that is algebraic or logical. Of course, many mixed approaches have been proposed making this classification questionable. A different classification can be found in (Wing, 1990a).

4.1. Descriptive Approaches Based on Algebraic Methods

Algebraic methods are based on the concepts of Abstract Data Type (ADT) (Guttag, 1977), (Guttag and Horning, 1978). With these methods of specification the system is described in an abstract manner; however, the description remains quite intuitive and lightly operational to be easily understandable. Most of the algebraic methods allow to specify the system at different levels of abstraction, starting from a coarse description and arriving at the most detailed one. For these methods, the system itself is regarded as an ADT, and its specification consists in describing its syntax and semantics. The syntax definition gives the description of the operator domains of the ADT, while the semantics is given by an implementation of these operators by means of mathematical expressions. Semantics is often defined by writing a set of axioms with a programming language based on first-order logic. Complex abstract data types are defined on the basis of simpler ones; hence, the semantics of complex types is specified by using the axioms of simple types, and thus the behavior of complex types can be again validated by using the axioms of the simple types. This allows to verify the specification correctness at each level of specification detail.

By iterating the ADT implementations the entire system is specified. Iteration ends when the elementary data types of the system are defined. Therefore, the system obtained is specified on the basis of few elementary ADTs, whose operators must be implemented by means of a traditional programming language (e.g., Pascal, C, etc.). The validation process is carried out with respect to high-level system properties. Then, if elementary ADTs are correctly implemented, the overall system will also be correct.

In the 1980s, many interesting specification languages have been proposed, according to the concept of ADT — e.g., ACT ONE (Algebraic Specification Technique (Ehrig and Mahr, 1985), which inspired LOTOS (Bolognesi and Brinksma, 1987), see Section 4.1.2), AFFIRM (Musser, 1980), and the Larch family of languages (Guttag, Horning and Wing, 1985).

Algebraic methods have been used for defining abstract data types in conventional applications (Musser, 1980); later on, they have been employed for specifying reactive systems and communication protocols (Sunshine, et. al, 1982). To give an idea of how these languages are structured, an example of protocol in AFFIRM (Sunshine, et. al, 1982) is reported in Fig.15, while Fig.16 shows the corresponding state machine in the Mealy

model (see Section 3.1). As can be noted, state variables are modeled by means of axioms, which in turn are functions of the axioms of other ADTs. For example, in Fig.16 the *SimpleMessageSystem* is defined by using *Message* and *QueueOfMessage* ADTs. For *QueueOfMessage* the operations of *NewQueueOfMessage*, *Add*, and *Remove* are defined. In general, the operators can be classified as constructors (*InitializeService, UserSend, SendComplete, UserReceive*), and selectors (*ReceiveComplete, Buffer, Sent, Received*). *State* is the axiom which models the data type behavior.

The completeness can be verified when it is proven that a defined property is verified by the axioms of the system. This confers a descriptive rather than the operational nature to these approaches, although the ADT behavior can be in many cases translated in state machines. The operational descriptions are distributed among the operators and, therefore, they are not simply executable. The property of liveness can also be verified, for example by proving that a message transmitted will be received in any case by the *SimpleMessageSystem*. In AFFIRM, there is no method for describing timing constraints.

The Larch family of specification languages has been defined on the basis of a common support, the so-called *used traits*. This support describes the common Larch model by means of an algebraic language — i.e., the *Larch Shared Language* (Guttag, Horning and Wing, 1985), (Garland, Guttag and Horning, 1990). By using this language new ADTs can be defined. An interface support must be defined on the Larch Shared Language by using a predicative language (e.g., pre- and post-conditions) (Wing, 1987). This layer plays the role of a support for a host language. For example, the Larch/Pascal provides a support for programming in Larch style by using the conventions of Pascal. On the contrary, each Larch language is based on the same support (i.e., Larch Shared Language). In the literature, there are many other Larch languages: the Larch/CLU (for CLU see (Liskov, et. al, 1977), (Liskov and Guttag, 1986)), Larch/ADA (Guaspari, Marceau and Polak, 1990), Larch/C, and also object-oriented languages such as the Larch/Smalltalk, the Larch/Modula-3 and the Larch/C++ (Wing, 1990b), (Leavens and Cheon, 1992), (Cheon and Leveson, 1993).

The above mentioned languages are enough formal to create specifications that can be easily verified, but unfortunately most of them are not supported by any specific construct for specifying timing constraints such as timeouts, deadlines, etc.

4.1.1. Z

The Z language is based on the theory of sets and predicate calculus (Abrial, 1982), (Sufrin, 1986), and was introduced in 1982. Differently from AFFIRM, the operations on a described data type are given by using the predicate logic (Spivey, 1988). As in other algebraic approaches, in Z the final specification is reached by refinement, starting from the most abstract aspects of the system. In Z, there is also a mechanism for system decomposition known as Schema Calculus. Therefore, a system specification is decomposed in smaller pieces called schemes where both static and dynamic aspects of system behavior are described.

```
type SimpleMessageSystem;
  needs types Message, QueueOfMessage;
  declare s: SimpleMessageSystem, m: Message;
  interfaces
    State(s): {ReadyToSend,Sending,ReadyToReceive,Acking};
    Sent(s), Received(s), Buffer(s): QueueOfMessage;
    InitializeService(s), UserSend(s,m), SendComplete(s): SimpleMessageSystem;
    UserReceive(s), ReceiveComplete(s): SimpleMessageSystem;
  axioms
    State (UserSend(s,m)) = if State(s) = ReadyToSend
                            then Sending
                            else State(s),
    State (SendComplete(s,m)) = if State(s) = Sending
                            then ReadyToReceive
                            else State(s),
    State (UserReceive(s)) = if State(s) = ReadyToReceive
                            then Acking
                            else State(s),
    State (ReceiveComplete(s)) = if State(s) = Acking
                            then ReadyToSend
                            else State(s),
    State (InitializeService) = ReadyToSend,
    Sent (UserSend(s,m)) = if State(s) = ReadyToSend
                            then Sent(s) Add m,
                            else Sent(s),
    Sent (InitializeService) = NewQueueOfMessage,
    Receive (UserReceive(s)) = if State(s) = ReadyToReceive
                            then Received(s) Add Front(Buffer(s))
                            else Received(s),
    Received (InitializeService) = NewQueueOfMessage,
    Buffer (UserSend(s,m)) = if State(s) = ReadyToSend
                            then Buffer(s) Add m
                            else Buffer(s),
    Buffer (ReceiveComplete(s)) = if State(s) = Acking
                            then Remove(Buffer(s))
                            else Buffer(s),
    Buffer (InitializeService) = NewQueueOfMessage,
end;
```

Figure 15. Example of *SimpleMessageSystem* in AFFIRM.

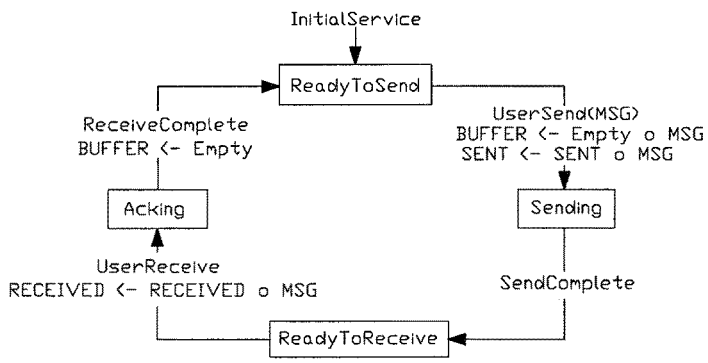

Figure 16. The corresponding state diagram of the *SimpleMessageSystem* as previously defined in AFFIRM.

A well-known specification example of the Z language is the Birthday book (Spivey, 1988), that is a system for recording people's birthday. Each entry contains a NAME and the DATE of birthday. The first step of specification consists in defining the state space of the system by means of the following schema:

BirthdayBook

$$known : \mathcal{P}\ NAME$$
$$birthday : NAME \mapsto DATE$$
$$known = dom\ birthday$$

specifying that the state space must satisfy the condition that *known* is equal to the domain of birthday and that the NAME is a known domain. It should be noted that in this example one person can have only one birthday, but the same birthday can belong to more than one person. Operations are defined by using other schemes, for example the schema *AddBirthdayBook* for adding a new element is reported in Fig.17.
In the *AddBirthdayBook* scheme, the qualifier Δ defines that the operation described can change the state space of *BirthdayBook*. The two declarations: *name? : NAME* and *date? : DATE* define that these are inputs (outputs are represented by the symbol "!" as in the CSP (Hoare, 1985)), while $name? \notin known$ imposes that the name read must not be already included in the BirthdayBook. The last line simply describes the update operation on the birthday set ($birthday'$ is the updated value). Schemes can be combined by means of the Z schema calculus, in which operations of *and*, *or*, etc., are defined. The Z language also includes a mathematical tool-kit allowing the definition of operations on sets and data. It is very useful for describing the mathematical aspects of a problem.

A specification in Z is a mixture of: informal text, definitions, axiomatic descriptions, constraints, type definitions, and schemes. Therefore, it cannot be considered a fully descriptive approach (Zave, 1990).

As regards real-time systems specification, the Z language does not have any support for defining timing constraints. Therefore, in the recent years, several extensions for adding time management have been proposed. In (Richardson, Aha and O'Malley, 1992), the Z language has been integrated with the RTIL (Real-Time Interval Logic) (Razouk and Gorlick, 1989).

Several object-oriented extensions of the Z language have been presented — e.g., OOZE (Alencar and Goguen, 1991), MooZ (Meira and Cavalcanti, 1991), Z++ (Lano,

AddBirthdayBook

$$\Delta\ BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{\ name? \rightarrow data?\ \}$$

Figure 17. Schema of AddBirthdayBook in Z language.

1991), Object-Z (Carrington, et. al, 1990). Most of them provide information hiding, inheritance, polymorphism and instantiation into the Z Schema Calculus. With these extensions, the system state space is defined as a composition of the state spaces of the individual system objects. The object-oriented paradigm has added formalism for modularity and specification reuse. Object-Z integrates also the concepts of temporal logic (Carrington, et. al, 1990), making it suitable for real-time specification. In this language the object status is a sort of event history of object behavior making the language more operational than the early version of Z.

4.1.2. LOTOS

LOTOS (Language Of Temporal Ordering Specification) is a formal technique belonging to the standard FDT defined within ISO (International Organization of Standardization) for the specification of open distributed systems (Bochmann, et. al, 1982). It was defined by ISO/TC97/SC21/WG1 subgroup C in 1981-86 (Bolognesi and Brinksma, 1987), (Bolognesi and Brinksma, 1989). LOTOS is based on the algebraic descriptive technique firstly presented by Milner (CCS) (Milner, 1980) and the abstract data type language ACT ONE (Ehrig and Mahr, 1985). Being based on ADT, LOTOS allows to define new ADTs. LOTOS uses the concepts of ADT for defining also the structural aspects of the system under specification, differently from other FDT which do not address the structural aspects.

LOTOS is strongly based on the concept of process. Structural decomposition is made on the basis of processes, and a distributed system is also regarded as a process with subprocesses. Algebraic operators are used to define relationships among processes — e.g., sequential ($\gg$), and parallel ($\|$) executions. For these reasons, LOTOS can be considered a process-oriented descriptive algebraic language. Processes communicate by means of messages, through gates. Messages can bring data or controls; they are considered events, are assumed to be atomic, and their occurrence is supposed to be instantaneous without time consumption. A system specification consists in the definition of process behavior, by describing how processes communicate, execute, and synchronize. Process definition specifies the temporal ordering in which a process interacts with other processes, by means of its gates. A process definition may include the definition of a set of types which are equivalent to ADTs.

Consistency among descriptions is verified by a syntax checker and by simulation. A compiler translating LOTOS specifications into a machine-oriented language is also available. The LOTOS tool has been produced by the ESPRIT project SEDOS. The G-LOTOS, which is a graphic editor to produce LOTOS specifications by means of a visual language is also present (Bolognesi, Najm and Tilanus, 1993). LOTOSPHERE is an integrated tool environment for defining systems in LOTOS (LOTOSPHERE, 1992). LOTOSPHERE is the result of an ESPRIT project (n.2304). With this tool the user can define process behavior in both descriptive and operational manner. The latter resort to the formalism based on extended finite state machines.

4.1.3. VDM

VDM (Vienna Development Method) dates back to the 1970s and to the work of a reasearch group at the IBM Laboratory in Vienna, attempting to create a formal approach capable of defining the programming language PL/I. Afterwards the original group was dispersed, but the ideas instead of dying spread to a larger community. The final outcome is reported in (Jones, 1986). At present, VDM is very popular and has become a British standard. VDM is mainly a specification language, but it can be profitably used for program designing and developing. Its mathematical support is used to verify the correctness of the resulting program by proving properties (Andrews, 1992).

The mathematical bases of VDM are the theory of sets and the theory of logic predicates. A VDM system specification consists in defining types, functions and operations, in the syntax of the so-called Meta-IV language. Data types can be defined by homogeneous or heterogeneous combinations of VDM basic types (natural numbers, integer, Boolean, etc.). For the new types, a set of operations (i.e., sum, etc.) is automatically available. Functions are defined as procedures which have as arguments, and return as results, elements of primitives or user-defined data types. Functions can also be specified through their pre- and post-conditions. An operation is applicable to a set of states selected on the basis of a pre-condition associated with the operation itself (thus it seems to be very similar to the concept of condition on transition). Operations can contain *read* and *write* of external events. A post-condition is also associated with an operation. The post-condition describes the state domain after the operation execution. The specification of a system is generated by starting from a coarse description, until the final specification is obtained by refinement (Fields and Elvang-Goransson, 1992). Specification consistency is verified by checking if the definitions at different levels of abstractions are consistent. The validation consists in proving if some selected important properties are verified by the given specification.

The VDM model has no mechanism for defining the system structure. Data types are defined in terms of other data types, without partitioning the system into communicating subsystems. However, the formalism is powerful enough to describe (with a certain effort) even these conditions, but the reuse of VDM specification, as for other ADT-based approaches, is very hard. VDM is widely employed for specifying safety critical systems by using specific extensions for managing timing constraints.

Recently, an object-oriented extension of VDM has been presented — e.g., VDM++ (Dürr and vanKatwijk, 1992). This supports inheritance and instantiation without allowing the mechanism related to polymorphism. VDM++ permits the definition of timing constraints which makes it adequate for real-time system specification.

4.2. Descriptive Approaches Based on Logical Methods

These methods describe the system under specification by means of a set of logic rules, specifying how the system must evolve from certain conditions. Differently from the operational methods, the state space described by these specifications is limited and abstract. Rules can be given in the form of first-order clauses of Horn or higher-order logical ex-

pressions (Maier and Warren, 1988). These languages are unsuitable for representing the structural aspect of a system, but are very appropriate for describing properties of the system under specification.

Validation consists in proving high-level properties, which are also given in the form of logical expressions, by means of theorem solvers or Prolog engines. Simulation is also based on the same techniques. For this reason, time of execution and time ordering of events during the proof can be unpredictable and thus the real-time execution of logical specifications is almost unfeasible.

Of course logic and temporal logic date back to the ancient Greeks. These have been brought in computer science in the 1970s (Gotzhein, 1992). During the 1980s some papers have been published dealing with the use of temporal logic for program specification (Schwartz and Melliar-Smith, 1982), (Jahanian and Mok, 1986). In the literature, there are many examples of logic languages for the specification of relationships among times and actions. These are often integrated with other techniques addressing also the functional and/or the structural aspects of the system under specification — e.g., RT-ASLAN (Real-Time extension of ASLAN) integrates the first-order logic with the ADT (Auernheimer and Kemmerer, 1986).

4.2.1. RTL

RTL (Real-Time Logic) is a formal language to describe the temporal relationships among events and actions (Jahanian and Mok, 1986). In RTL, the concept of time is absolute and the execution semantics quite independent of the scheduling mechanism, since all the language constructs are defined in terms of the symbol @, which assigns the current value of time to event occurrence.

In RTL, there are three types of constants, that is, actions, events, and integers. Actions can be simple or composite: the latter can be sequential or concurrent. In turn, events are divided in three classes: start/stop, transition, and external. Events and actions are similar to stimuli and responses, respectively, as defined by Dasarathy (Dasarathy, 1985). Periodic events are specified through recursive predicates. Integers can be either time durations or number of events. A system specification in RTL consists in deriving a set of axioms from the event-action model of the system, considering: (a) the relations between events and their 'start' and 'stop' occurrences; (b) periodic or sporadic events; (c) causes of transition; and (d) artificial constraints on the internal behavior. A system property (i.e., an RTL assertion) can be proven by refutation. For example, considering the specification reported in (Jahanian and Mok, 1986): "Upon pressing button #1, action SAMPLE is executed within 30 time units. During each execution of this action, the information is sampled and sequentially transmitted to the display panel. The computation time of action SAMPLE is 20 time units.", its translation in RTL results to be:

$$\begin{aligned} &\forall x : @(\Omega button1, x) \leq @(\uparrow SAMPLE, x) \wedge \\ &\quad @(\downarrow SAMPLE, x) \leq @(\Omega button1, x) + 30 \end{aligned} \tag{1}$$

$$\forall y : @(\uparrow SAMPLE, y) + 20 \leq @(\downarrow SAMPLE, y)$$

where Ω means that the variable corresponds to an external event. The so-called *Constraint Graph* is constructed from the RTL specification and is used to verify the safety of the system. The constraint graph simplification by means of simple rules permits the detection of incongruences among temporal constraints. RTL is supported by an automatic inference procedure to perform reasoning about timing properties. RTL specifications can be generated directly from a description given by using the notation of Modecharts (Jahanian and Stuart, 1988).

4.2.2. TRIO

TRIO (Tempo Reale ImplicitO) is a language based on first-order logic, augmented by temporal operators (Ghezzi, Mandrioli, and Morzenti, 1990). It allows to define logic equations which may include timing relationships. A time-dependent TRIO formula is given with respect to the current time; time is implicit. Temporal relationships between events are expressed on the basis of the operator *Dist(F,t)* (Mandrioli, Morasca and Morzenti, 1992, (Felder, Mandrioli, and Morzenti, 1991), which is satisfied at the current time if and only if the property F holds at an instant which is distant t time units from the current time. Many other operators are defined for describing system behavior in TRIO logical expressions in the past and in the future. For example, the special functions $Futr(F,t) = t \geq 0 \wedge Dist(F,t)$, and $Past(F,t) = t \geq 0 \wedge Dist(F,-t)$ are defined.

Since TRIO is based on a completely formal syntax and semantics, and includes the managing of time, it is intrinsically executable, in the sense that from a TRIO formula a precise model can be generated in which the variables inside the predicates have well-defined values. Expressions are usually given in the implicative forms:

$$A \rightarrow B \stackrel{def}{=} \neg A \vee B$$

$$A \leftrightarrow B \stackrel{def}{=} (A \rightarrow B) \wedge (B \rightarrow A)$$

where $\neg$ is the *not*, $\vee$ the *or* and $\wedge$ the *and* Boolean operator. *For all* $\forall$ and the *existence* (i.e., $\exists x A \stackrel{def}{=} \neg \forall x \neg A$) qualifier can be used.

The following example is quoted from (Mandrioli, Morasca and Morzenti, 1992). Consider a pondage power station where the quantity of water held in the tank is controlled by means of a sluice gate. The gate is controlled by the commands: *up* and *down* which respectively open and close the gate. These are represented by a predicate *go* having a range { *up, down* }. The gate can be in one of the states: *up, down, mvUp, mvDown*. The state is modeled by a time-dependent variable named *position*. The following formula describes that the gate in Δ time units passed from the *down* to the *up* condition after receiving a *go(up)* command:

$$(\ position{=}down \wedge go(up)) \rightarrow (Lasts(position{=}mvUp,\ \Delta) \wedge Futr(position{=}up,\Delta))$$

When a *go(up)* command reaches the system and the gate is not yet in the *down* position, but it is moving down for a previous command *go(down)*, then the direction of motion is not changed but the system waits until the *down* position is reached:

$$position{=}mvDown \wedge go(up) \rightarrow$$
$$\exists\, t\, (NextTime(position{=}down,\, t) \wedge Futr(\, Lasts(position{=}mvUp, \Delta) \wedge Futr(position{=}up, \Delta),\, t\,)\,)$$

where $NextTime(F,d){=}Futr(F,t) \wedge Lasts(\neg F,t)$, and $Lasts(F,t){=}\forall t'\ (0 < t' < t \rightarrow Dist\ (F,t))$. Since the gate behavior can be supposed to be symmetric with respect to its direction of motion, other two similar expressions should be written which describe the commands and their effects.

A TRIO specification can be validated against high-level properties described by means of the same formalism. Moreover, an efficient interpreter is available that makes a TRIO specification executable for real-time systems. Since the time relationships are given implicitly and the time is expressed in time units, the absolute time constraints cannot be specified. On the other hand, it has the capability to guaranteeing system safety by verifying the temporal ordering among events, independently of the underlying hardware.

4.2.3. Other Descriptive Approaches Based on Logical Methods

Many other interesting approaches, which essentially correspond to extensions of temporal logic — e.g., CTL (Computation Tree Logic) (Emerson and Halpern, 1986), RTIL (Real-Time Interval Logic) (Razouk and Gorlick, 1989), TCTL (Timed CTL) (Alur, 1990), TPTL (Timed Propositional Temporal Logic) (Alur and Henzinger, 1990), have been defined. Most of these approaches do not cover the structural and functional aspects of the system under specification.

5. Dual Approaches

In order to obtain the best benefits from the descriptive and the operational approaches, in the late 1980s the so-called "dual approaches" have begun to appear (Ostroff, 1989), (Felder, Mandrioli, and Morzenti, 1991). Dual methods try to integrate in a single approach the formal verifiability of descriptive approaches and the executability of operational approaches, though they are often in contrast, especially as regards the reuse and the verification of software specifications,

In effect, an ideal tool for specifying real-time systems should be:

1. An *easy and intuitable method and tool.* Where, "easy" means "very close" to the analyst mindset. For this reason, the tool must be endowed with a graphic user interface, and it must allow both top-down and bottom-up approaches for software specification, as well as a combination of these. Operational models seem to be suitable for this purpose, since they can be visually represented; in the literature, there are many examples of their use in both top-down and bottom-up approaches for software specification.

2. A model to make easier the *reusing of reactive system specifications.* This means that the model adopted must provide support for software composition by reusing already defined software components. In the specification of reactive systems, both

static (module interface and structure) and dynamic aspects (module behavior with timing constraints) should be reused. In addition, since the system specification must be validated to ensure its correctness, also the process of composition/decomposition must be supported by a validation method technique. For this purpose, descriptive formal methods are strongly preferable with respect to operational methods, since the validation of decomposition cannot be performed through simulation but only by means of proof of properties. This is related to the fact that during the decomposition low-level details are not yet available.

3. A *method for verifying and validating the specified software against critical conditions since the early phases of system specification.* This feature with that of the previous point should allow the verification and validation at each level of abstraction even if the implementation details are not available (such as in the early phases of system specification — i.e., partial specification). It should be noted that the operational models, differently from the descriptive ones, are not suitable to be executed when the model is partially specified. For this reason, descriptive methods seem to be preferable for this purpose, even if with these methods the validation is usually carried out through properties proof.

4. An *executable model to allow the validation of system behavior by means of simulation.* The simulation of an executable model improves the confidence of system validation and, together with the above features, provides support for rapid system prototyping. For this purpose, operational models could be profitably used, while most of the descriptive models are not efficient since they are usually "executable" by means of inferential engines which are typically strongly inefficient.

As has been pointed out, the above objectives are often in contrast, and they cannot be met by a specification approach which is only descriptive or only operational. Recently, several dual methods to overcome these difficulties have been defined (Bucci, et. al, 1993), (Mandrioli, 1992).

To a certain extent, several of the already discussed approaches can be considered dual languages. On the contrary, only those that have both operational and descriptive semantics allowing the specification executability and the verification of properties should be considered really dual. One of the first examples of dual approach can be considered the Transition Axiom Method proposed by Lamport (Lamport, 1993), (Lamport, 1989). In this method, the specification is equivalent to a state machine, on which the proof of high-level properties given by means of axioms can be verified.

5.1. ESM/RTTL

ESM/RTTL is a dual approach obtained be the integration of ESM (Extended State Machine) language and RTTL (Real-Time Temporal Logic) (Ostroff and Wonham, 1987), (Ostroff, 1989).

ESM is an operational model based on communicating finite state machines in which variables with arbitrary domains are used (Ostroff and Wonham, 1987). The *operations*

allowed are assignments, send or receive. The state machine follows a Mealy model in which conditions on transitions (in ESM, they are called *guards*) between states are equivalent to first-order expressions on state variables, while the output is an assignment to state variables. Each event is represented by an exit activity A_e, a source activity A_s, an *operation* and a *guard*: $(A_e, guard, operation, A_s)$. A system description refers to only a single state domain. The concept of time is enforced by means of a global time variable which can be tested, updated, and increased or not at each state transition. Time is discrete, and a state transition can be formally executed in zero time units. For each state transition the minimum and the maximum time can be specified in which the enabling condition becomes true.

RTTL is a logic language based on the classical operator of temporal logic: *until* ($\sqcap$), and *next* ($\odot$). From these the more useful operators of: eventually ($\Diamond$), henceforth ($\Box$), etc., are derived. RTTL can be used to describe high-level properties of the system under specification by means of first-order logic formulae.

The integration between RTTL and ESM is obtained by describing the high-level behavior of the system with first-order expressions in which conditions for transitions containing RTTL expressions can be also present. Both RTTL and ESM formulae can refer to the absolute time value.

5.2. *TRIO+*

TRIO+ (TRIO object-oriented) is a logical language for modular system specification (Mandrioli, 1992), (Mandrioli, 1993) extending TRIO (see Section 4.2.2) with object-oriented capabilities. It is based on a first-order temporal language, providing support for a variety of validation activities, such as testing, simulation and property proof. TRIO+ is considered a dual language since it combines the use of visual notation, hierarchical decomposition (typically of operational approaches), with the rigour of the descriptive logical language. In Fig.18, the example reported in Section 4.2.2 for the TRIO language has been rebuilt in TRIO+. Since TRIO+ is based on logic programming, the object-oriented concept of an instance corresponds to a history of the Prolog interpreter (that is the history of the status of an object).

Differently from TRIO, TRIO+ is endowed with a graphical notation that covers only the declarative part of the language. With this graphic interface the structural aspects can be described, by defining the components of a class and their relationships (see Fig.19). TRIO+ is an executable model which supports the executions of partially defined specifications.

5.3. *TROL*

TROL (Tempo Reale Object-oriented Language) is an object-oriented dual language for the specification of real-time systems (Bucci, et. al, 1994). TROL adopts a dual model which is able to satisfy the above requirements presenting both descriptive and operational aspects. TROL adopts a modified object-oriented model, and has the capability to

Class sluice_gate
  visible go, position
  temporal domain integer
  TD Items
    Predicates go({up,down})
    vars position: { up, down, mvup, mvdown }
  TI Items
    vars Δ : integer
  axioms
    vars t: integer
    go_down: position=up $\wedge$ go(down) $\rightarrow$ Lasts(position=mvdown,Δ) $\wedge$ Futr(position=down,Δ)
    gp_up: position=down $\wedge$ go(up) $\rightarrow$ Lasts(position=mvup,Δ) $\wedge$ Futr(position=up,Δ)
    move_up: position=mvup $\wedge$ go(down) $\rightarrow$ $\exists$t (NextTime(position=up,t) $\wedge$
        Futr(Lasts(position=mvdown,Δ) $\wedge$ Futr(position=down,Δ),t)
    move_down: position=mvdown $\wedge$ go(up) $\rightarrow$ $\exists$t (NextTime(position=down,t) $\wedge$
        Futr(Lasts(position=mvup,Δ) $\wedge$ Futr(position=up,Δ),t)
end sluice_gate

Figure 18. Textual description of the class *sluice_gate* in TRIO+.

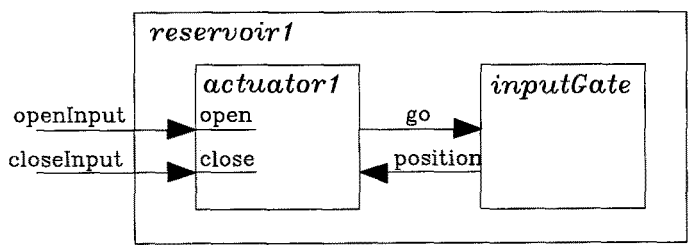

Figure 19. Visual description of class *reservoir1* comprised of *actuator1* and *inputGate* objects in TRIO+.

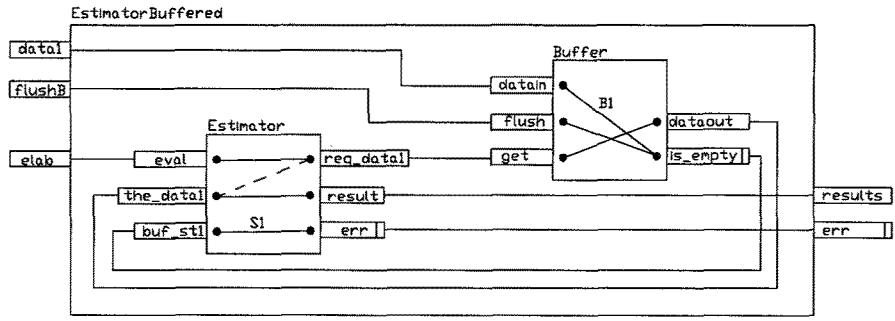

Figure 20. The class EstimatorBuffered as comprised of S1 and B1 sub-objects (i.e., its attributes) in TROL.

describe the system behavior, its functionality and structural aspects. Moreover, differently from other dual models and languages, the TROL model is mainly based on reusing both static and dynamic aspects of specifications. TROL is based on an extension of timed CSP — i.e., the CRSM (Shaw, 1992) (see Section 3.1.5).

In TROL, the system under specification is hierarchically decomposed in objects and sub-objects. For these objects, the behavior can be specified by means of first-order clauses, since the early phases of decomposition (see Fig.20, and Fig.21). Moreover, also timing constraints at the external interface of each object can be defined according to (Dasarathy, 1985). These constraints can be associated with *Provided* and *Required* services of each class, and to *Clauses*. TROL allows to describe the system at different levels of structural abstractions and of specification details without boundaries among the specification steps. The TROL model allows the verification and validation of composition/decomposition mechanisms. At each specification level, TROL helps the user in the verification of consistency, thus allowing the incremental specification and the execution of partially specified systems (i.e., prototyping) (Bucci, et. al, 1993). These features are very useful when a component under reuse can be verified and validated in order to check if it satisfies the requirements.

Objects that cannot be further decomposed are defined as extended state machines (see Fig.22a and 22b). These are internally concurrent, defining in this way a high reactive architecture. The state machine model supports the definition of timing constraints such as timeout, and minimum and maximum time for transition. Moreover, any time failure can be recovered by using special functions.

In TROL, the descriptive aspects of the language are used to help the developer to generate a correct, complete and congruent specification, validating the system composition/decomposition by means of clauses and the reasoning on timing constraints. Also the state machines are validated by using clauses. Thus, the final validated model is executable by using the operational model of state machines. Is should be noted that in TROL the analysts use a descriptive language in the phase of analysis while state machines are used in the phase of design. TROL supports all the aspects of the object-oriented paradigm allowing inheritance, instantiation, etc. In order to guarantee the predictability and, hence, an a-priori real-time schedulability, some assumptions have

```
Class Estimator specializing XSM
    Provided_services:
        eval : Signal;
        the_data1 : DataType;
        buf_st1 : Boolean;
    Required_services:
        req_data1 : Signal;
        result : Real;
        err available : EstimatorErrType;
    Clauses:
        REQ_DATA1: New(eval) ∧ err==OK → Ready (req_data1);
        WAITDATA1: reverse Ready(req_data1) → New(the_data1);
        RESULT: New(the_data1) → Ready(result);
        BUFEMPTY: buf_st1 → err==EMPTY ;
end;

Class EstimatorBuffered specializing non_basic_object_class
    Provided_services:
        data1 : DataType;
        flushB : Signal;
        elab [4,6] : Signal;
    Required_services:
        results : Real;
        err available : EstimatorErrType;
    Clauses:
        ESTIMATION: New(elab) ∧ err==OK → Ready(results) - - [2, 3.1];
        FLUSH : New (flushB) → err==EMPTY;
        DATA : New (data1) ∧ err==EMPTY → err==OK;
    /*** private parts ***/
    Attributes:
        B1 : Buffer;
        S1 : Estimator;
    Connections:
        data1 - - B1.datain;      elab - - S1.eval;
        S1.result - - results;    S1.err - - err;
        S1.req_data1 - - B1.get;      B1.dataout - - S1.the_data1;
        B1.is_empty - - S1.buf_st1;      flushB - - B1.flush;
end;
```

Figure 21. Description in TROL of class EstimatorBuffered with the external description of class Estimator, where EstimatorErrType is defined as an enumeration: **enum** EstimatorErrType {EMPTY,OK};.

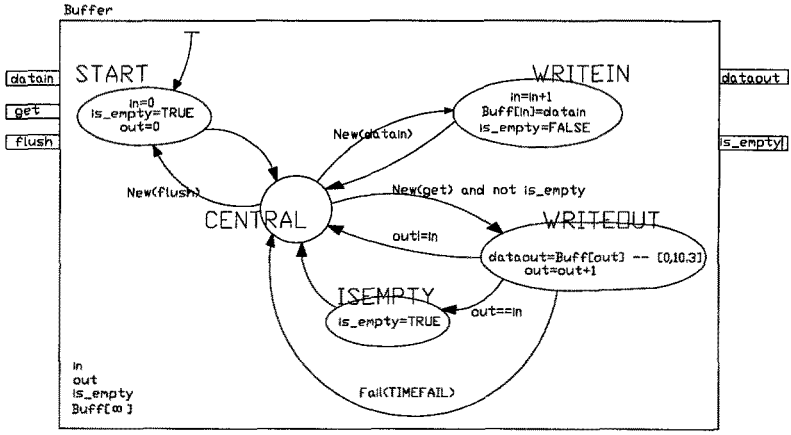

Figure 22a. Textual descriptions of class Buffer in TROL.

been made on the TROL model: no dynamic binding, no dynamic object instantiation (no dynamic sizing of object collections), no direct or indirect recursion in class specifications, no multiple inheritance among classes is allowed.

TROL has its visual representation which is supported by a CASE tool named TOOMS (Bucci, et. al, 1993). TOOMS consists in a set of visual editors, a report generator, a database for collecting and recovering specifications for reuse, a compiler, an analyzer to perform the verification of completeness and consistency, a simulator (that can simulate the system behavior by using both clauses and state machines) and a metricator (Bucci, et. al, 1993), (Campanai and Nesi, 1994). Automatic code generation is also provided through the TROL-compiler which transforms the TROL code in C++ for an *ad-hoc* real-time kernel called TROL-KERNEL working on OS/2 and UNIX.

6. Discussion and Conclusions

The most significative formal methods for the specification of real-time systems have been reviewed, with respect to the power of formalism, the tool capability, and the low-level characteristics, as discussed in the introduction. The presentation has been referred to the historical evolution.

In Tab.1 a summary of the main features of the tools analyzed is reported. The legend of the Table reported in the caption is self-explicative and thus we refrain from commenting on it. However, it is worth to spend a few words on columns labelled as *Sem.* (for semantics) and *Orien.* (for orientation). In the column *Sem.*, the approaches are classified as operational and/or descriptive, considering as dual those which present both these aspects. Column *Orien.*, indicates whether the approach is Process-, State-, Functional-, or sTructure-oriented. Process-oriented approaches are those which consider the system as decomposed in processes; state-oriented approaches are those that are

```
Class Buffer specializing XSM
    Provided_services:
        datain : DataType ;
        get : Signal ;
        flush : Signal ;
    Required_services:
        dataout : DataType ;
        is_empty available : Boolean;
    Clauses:
        GET : New (get ) ∧¬ is_empty → Ready (dataout );
        FLUSH : New (flush) → is_empty;
        DATAIN : New (datain) ∧ is_empty → ¬ is_empty;
    /*** private parts ***/
    Attributes:
        in : Integer ;
        out : Integer ;
        is_empty : Boolean ;
        Buff : DataType [∞ ];
    States:
        START: { in=0; is_empty=TRUE; out=0; }
        CENTRAL: { }
        WRITEIN: { in=in+1; Buff[in]=datain; is_empty=FALSE; }
        WRITEOUT: { dataout=Buff[out] - - [0,10.3]; out=out+1; }
        ISEMPTY: { is_empty=TRUE; }
    Paths:
        INIT: { START: → CENTRAL;
                CENTRAL: New(flush) → START;}
        PUT: { CENTRAL: New(datain) → WRITEIN;
               WRITEIN: →CENTRAL; }
        GET: { CENTRAL: New(get) ∧¬ is_empty → WRITEOUT;
               WRITEOUT: in != out → CENTRAL;
               WRITEOUT: in==out → ISEMPTY;
               ISEMPTY: → CENTRAL;
               WRITEOUT: Fail(TIMEFAIL) → CENTRAL; }
end;
```

Figure 22b. Visual descriptions of class Buffer.

focused on describing the system state domain; functional-oriented approaches are those that decompose the system in data transformations; and structure-oriented approaches are those that consider the system as a set of sub-systems (i.e., objects or modules). In this column only the two most relevant aspects are reported.

It can be noted that during the last 20 years most of the early approaches for describing system behavior (e.g., Z, VDM, SDL, etc.) have been integrated by using high-level methods in order to improve their capability in modeling all the system aspects (i.e., structural, behavioral and functional) and providing support for reusability. In many cases, this coverage has been reached by transforming the model from process- or state- or function-oriented to object-oriented (Z++, VDM++, OSDL, etc.). The adoption of the object-oriented paradigm has added to the early formalisms the capability of system structuring (by means of the concept of *class*) and of reusing real-time specifications (by means of the concepts of *inheritance*, *polymorphism*, and *instantiation*). This trend holds for both descriptive, and operational approaches as well as for dual approaches.

For most of the formal methods, facilities for managing timing constraints have also been added. Operational approaches, based on state machines and Petri nets have been extended so as to include the notion of time by augmenting the model with first-order logic. The resulting models are more complex to be verified and validated, but the verification and validation can be performed in the same manner under certain restrictive conditions.

The operational approaches based on other notations (firstly created as methodologies for supporting the analysis and/or design phases of the system under specification, e.g., DFD, JSD, Booch, Wirsf-Brock, etc.) (see Section 3.3) have followed a different path. The early versions of most of these methodologies have been integrated by using low-level methods for supporting the lack of formalities and for covering the design phase of software life-cycle. Currently, methods exist which are based on DFD and state machines (e.g., RT-DFD (Hatley and Pirbhai, 1987)), JSD and state machines (e.g., Entity-Life (Sanden, 1989c), (Sanden, 1989b)), DFD and Petri nets (e.g., IPTES (Pulli, et. al, 1991)), extended Entity-Relationships and VDM (e.g., ATMOSPHERE (Dick and Loubersac, 1991)), object-oriented model and state machines (e.g., Booch (Booch, 1991), Shlaer and Mellor (Shlaer and Mellor, 1991)), object-oriented model and Petri nets (e.g., PROTOB (Baldassari and Bruno, 1991)), etc.. For most of them, the definition and verification of timing constraints and the final validation are still a problem. For example, when the definition of timing constraints is allowed in the early phases of system specification, their consistency is not verified with respect to the low-level description. On the other hand, the definition of timing constraints is allowed only at the low-level — i.e., at the level of state machines or Petri nets, where the consistency can be verified and the validation performed. It should be noted that, in the latter case, timing constraints are specified when the system architecture is already defined and, therefore, their verification can lead to demonstrate that the system structure (e.g., decomposition) is wrong or partially incorrect.

Due to the fact that formal languages are too far from the analysts mindset to be easily adopted for specifying real-time systems (since they need too many details), CASE tools have been implemented including visual editors, compilers, metric support, configuration

Table 1. Summary of formalism evaluation, where: *Sem.* (semantics): Operational and/or Descriptive (Algebraic or Logic); *Orien.* (orientation): Process-, State-, Functional-, sTructure-oriented; *Desc.* (description): Textual (5pt.), Visual (5pt.); *Cov.* (coverage): Structural (3pt.), Behavioral (5pt.), Functional (2pt.); *Comm.* (communications among processes); *Time* (time model): Implicit (3pt.), Explicit (7pt.) (Relative or Absolute), None (0pt.); *Verif.* (verification of consistency and congruence): Yes (10pt.) or No (0pt.); *Valid.* (validation of system behavior): Static (6pt.) (i.e., by proving properties), Dynamic (4pt.) (i.e., by simulation); *Exec.* (executable specification, by means of interpretation or simulation): Yes (10pt.) or No (0pt.); *Prot.* (prototyping — i.e., simulation or execution of partial specifications): Yes (10pt.) or No (0pt.).

	Sem.	Orien.	Desc.	Cov.	Comm.	Time	Verif.	Valid.	Exec.	Prot.
PAISLey	O	P	T	B,F	A	ER	Y	S,D	Y	Y
SDL	O	P	V,T	S,B	A	ER	Y	D	Y	Y
OSDL	O	P	V,T	S,B,F	A	ER	Y	D	Y	Y
Esterel	O	P	T	B,F	S	EA	Y	S,D	Y	N
Statecharts	O	S	V	B	S	ER	Y	D	Y	N
Objectcharts	O	S	V	B,S	S	EA	Y	D	Y	N
RTRL	O	S	T	B	N	ER	Y	S	Y	N
CRSM	O	P,S	V,T	B	S	ER	Y	D	Y	N
PN	O	S	V	B	A	I	Y	D	Y	N
CmPN	O	P,S	V	S,B	A	I	Y	S,D	Y	N
SPN	O	S	V	B	A	ER	Y	S,D	Y	N
PROT nets	O	P,S	V,T	S,B	A	ER	Y	S,D	Y	Y
TPN	O	S	V	B	A	ER	Y	S,D	Y	N
Timed PN	O	S	V	B	A	ER	Y	S,D	Y	N
HMS	O	S	V	B,S	A	ER	Y	S,D	Y	Y
SADT	O	F,S	V	F,S,B	N	EA	Y	N	N	N
DFD	O	F	V	F,S	N	N	Y	N	N	N
RT-DFD	O	F,S	V	F,S,B	N	EA	Y	D	Y	Y
JSD	O	T	V	S	N	N	Y	N	N	N
Entity-Life	O	P,S	V	S,B	A	ER	Y	D	Y	Y
Wirsf-Brock	O	P,F	V	S,F	S	N	N	N	N	N
HOOD	O	T,S	V	S,B	A,S	EA	Y	N	N	N
BOOCH	O	P,S	V	S,B	A,S	EA	N	N	N	N
OMT	O	T,S	V	S,B,F	A	EA	N	N	N	N
Shlaer-Mellor	O	T,S	V	S,B	A	EA	N	N	N	N
Coad-Yourdon	O	T,F	V	S,F	A	EA	N	N	N	Y
AFFIRM	DA	S	T	B	N	N	Y	S	N	Y
Larch	DA	S	T	B	N	N	Y	S	N	Y
Larch/C++	DA	S	T	B	A,S	N	Y	S	N	Y
Z	DA	S	T	S,B	N	N	Y	S	N	Y
Object-Z	DA	S	T	B	S	ER	Y	S	Y	Y
LOTOS	DA	P,S	T	B,S	A	N	Y	S	Y	Y
G-LOTOS	DA	P,S	T,V	B,S	A	N	Y	S	Y	Y
VDM	DA	S	T	B	N	N	Y	S	N	Y
VDM++	DA	S	T	B	N	ER	Y	S	Y	Y
RTL	DL	S	T	B	N	ER	Y	S	N	N
Modecharts	DL	S	V,T	B,S	S	ER	Y	S,D	Y	N
TRIO	DL	S	T	B	N	I	Y	S,D	Y	Y
TCTL	DL	S	T	B	N	ER	Y	S	N	N
CTL	DL	S	T	B	N	N	Y	S	N	N
ESM/RTTL	O,DL	T	T	B	N	ER	Y	S,D	Y	N
TRIO+	O,DL	T,	T,V	B,S	N	I	Y	S,D	Y	Y
TROL	O,DL	T,S	T,V	B,S,F	S,A	ER	Y	S,D	Y	Y

management support, report generators, simulators, test generators, etc. A CASE tool with a visual interface makes easier the work of the designer by representing formal syntax through graphic symbols and, thus, the user is helped by collecting the specification details in a structured manner. Of course, a CASE tool must maintain consistency between the visual representation and the syntax and semantics of the model. As a result, by means of a CASE tool, a specification language improves its power. From this point of view, the operational approaches have an advantage over the descriptive approaches, since they are intrinsically endowed of a visual notation, while the definition of a visual language supporting the syntax of the latter is a more difficult task. In general, the presence of an integrated CASE tool gives a major confidence to the specification quality, improving the fulfilment of requirements, the verification of consistency, the validation of system behavior with respect to temporal constraints, etc.

A graph reporting the trend of the specification tools capability with respect to last 20 years is reported in Fig.23. This graph has been drawn on the basis of tools analysis carried out in this paper, considering a particular score for each feature. Scores have been defined on the basis of their usefulness in specifying real-time systems. Scores associated with the different features are reported in the caption of Tab.1. As is appeared from this graph, the number of positive features is increasing with time. This growth has been obtained in many cases by integrating different approaches, and thus transforming the early nature of a model towards a dual approach. In our opinion, in the next years, we will witness a tangible additional growth of tool capabilities These improvements will be mainly focused on tools integration, so as to help the analyst in all phases of the software life-cycle, without boundaries from one phase to another. The integrated CASE tools will give a major confidence for the specification of *perfect* software (e.g., a software which is safe, congruent, complete, satisfying temporal constraints, etc.).

Acknowledgments

The authors want to thank CESVIT (High-Tech Agency, Italy) which allowed them to test most of the tools mentioned in this paper (StP of Interactive Development Environments, ARTIFEX of ARTIS, GEODE of Verilog, Teamwork of CADRE, EXCELERATOR of Index Technology, etc.). In addition, they wish to thank also A. Corgiatini, R. Mattolini, O. Morales, M. Traversi, and E. Vicario, for their help.

References

Jean-Raymond Abrial. The Specification Language Z: Basic Library. Technical report, Programming Research Group, Oxford University, UK, 1982.

G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, London, 1986.

A. J. Alencar and J. A. Goguen. OOZE: An Object Oriented Z Environment. In *Proc. of European Conference on Object Oriented Programming, ECOOP'91*, pages 180–199. Springer Verlag, Lecture Notes in Computer Sciences, LNCS n.512, 1991.

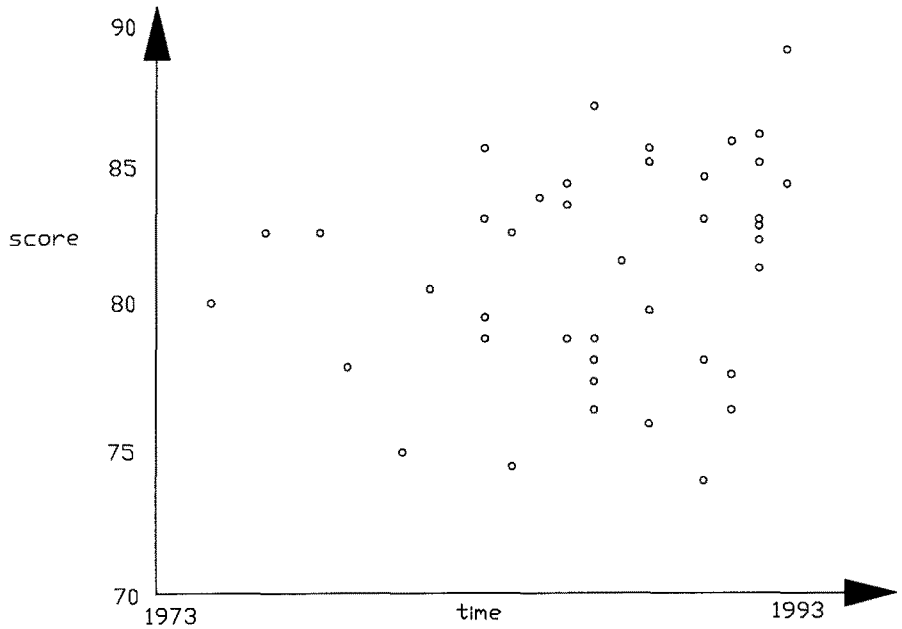

Figure 23. Trend of tool capabilities in modeling specifications for real-time systems in the last 20 years.

M. Alford. A Requirements Engineering Methodology for Real-Time Processing Requirements. *IEEE Transactions on Software Engineering*, 3(1), Jan. 1977.

M. Alford. SREM at the Age of Eight; The Distributed Computing Design System. *Computer*, April 1985.

R. Alur, C. Coucorbertis, and D. Dill. Model-Checking for Real-Time Systems. In *Proc. of 5th IEEE LICS 90*, pages 414–425. IEEE, 1990.

R. Alur and T. A. Henzinger. Real Time Logics: complexity and Expressiveness. In *Proc. of 5th IEEE LICS 90*. IEEE, 1990.

D. Andrews. VDM Specification Language, Proto-Standard. Technical report, BSI IST/5/50, Leicester University, 1992.

ARTIFEX. ARTIFEX User's Manual, ver.3.0. Technical report, ARTIS, Turin, Italy, 1993.

B. Auernheimer and R. A. Kemmerer. RT-ASLAN: A Specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):879–889, Sept. 1986.

M. Baldassari and G. Bruno. PROTOB: an Object Oriented Methodology for Developing Discrete Event Dynamic Systems. *Computer Languages,*, 16(1):39–63, 1991.

H. Barringer. *A Survey of Verification Techniques for Parallel Programs.* Lecture Notes in Computer Science 191, Springer Verlag, New York, 1985.

Z. Bavel. *Introduction to the Theory of Automata.* Reston Publishing Company, Prentice-Hall, Reston, Virginia, 1983.

G. Berry and L. Cosserat. *The ESTEREL Synchronous Programming Language and Its Mathematical Semantics.* Springer Verlag, Lecture Notes in Computer Science, LNCS n.197, 1985.

B. Berthomieu and M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

T. E. Bihari and P. Gopinath. Object-Oriented Real-Time Systems: Concepts and Examples. *Computer*, pages 25–32, Dec. 1992.

N. D. Birrel and M. A. Ould. *A practical handbook for software development.* Cambridge University Press, Cambridge U.K., 1985.

G. V. Bochmann, E. Cerny, M. Cagne, C. Jard, A. Leveille, C. Lacaille, M. Maksud, K. S. Raghunathan, and B. Sarikaya. Experience with Formal Specification Using an Extended State Transition Model. *IEEE Transactions on Communications*, 30(12):2505–2513, Dec. 1982.

T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P. H. J. van Eijk and C. A. Vissers, editors, *The Formal Description Technique LOTOS*, pages 23–71. Elsevier Science Publisher, North-Holland, 1989.

T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, 1987.

T. Bolognesi, E. Najm, and P. A. J. Tilanus. G-LOTOS: a Graphical Language for Concurrent Systems. Technical report, CNR Istituto CNUCE, PISA, Italy, March 15 1993.

G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211–221, Feb. 1986.

G. Booch. *Object-Oriented Design with Application*. The Benjamin/Cummings Publishing Company, California, USA, 1991.

T. L. Booth. *Sequential Machines and Automata Theory*. John Wiley and Sons, New York, USA, 1967.

R. Braek and O. Haugen. *Engineering Real Time Systems: An object-oriented methodology using SDL*. Prentice hall, New York, London, 1993.

P. Brinch-Hansen. The programming language concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

S. D. Brookes. On The Relationship of CCS and CSP. In J. Diaz, editor, *Automata, Language and Programming, Proc. of 10th Colloquium*, pages 83–96, Barcelona, Spain, July 1983. Springer Verlag, Lecture Notes in Computer Science, LNCS 154.

G. Bruno and G. Marchetto. Process-translatable Petri nets for the rapid Prototyping of process control systems. *IEEE Transactions on Software Engineering*, 12(2):346–357, Feb. 1986.

G. Bucci, M. Campanai, P. Nesi, and M. Traversi. An Object-Oriented CASE Tool for Reactive System Specification. In *Proc. of 6th International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE)*, Le CNIT, Paris la Defense, France, 15-19 Nov. 1993.

G. Bucci, M. Campanai, P. Nesi, and M. Traversi. An Object-Oriented Dual Language for Specifying Reactive Systems. In *Proc. of IEEE International Conference on Requirements Engineering, ICRE'94*, Colorado Spring, Colorado, USA, 18-22 April 1994.

G. Bucci, R. Mattolini, and E. Vicario. A Framework for the Development of Distributed Object-Oriented Systems. In *Proc. of the International Symposium on Automated and Decentralized Systems, ISADS'93*, pages 44–51. IEEE Press., Kawasaky, Japan, March 1993.

S. Budkowski and P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

J. R. Cameron. An Overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222–240, Feb. 1986.

M. Campanai and P. Nesi. Supporting Object-Oriented Design with Metrics. In *Proc. of the International Conference on Technology of Object-Oriented languages and Systems, TOOLS Europe'94*, Versailles, France, 7-11 March 1994.

D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In S. T. Voung, editor, *Formal Description Techniques*. Elsevier Science, 1990.

P. P. Chen. The Entity Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9, March 1976.

Y. Cheon and G. T. Leveson. A Quick Overview of Larch/C++. Technical report, Dept. of Computer Science, Atanasoff Hall Iowa State University, Ames, Iowa 50011-1040, USA, March 1993.

P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, New Jersey, USA, 1991.

D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, 18(1):9–18, Jan. 1992.

B. Cox. Message/Object Programming: an Evolutionary Change in Programming Technology. *IEEE Software*, 1(1):50–61, 1984.

B. Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them. *IEEE Transactions on Software Engineering*, 11(1):80–86, Jan. 1985.

T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, Prentice Hall, 1979.

J. Dick and J. Loubersac. Integrating Structured and Formal Methods: A Visual Approach to VDM. In A. vanLamsweerde and A. Fuggetta, editors, *Proc. of 3rd European Software Engineering Conference,*

ESEC91, pages 37–59, Milan, Italy, Oct. 1991. Springer Verlag, Lecture Notes in Computer Sciences, LNCS 550.

J. Diederich and J. Milton. Object, Message, and Rules in Database Design. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts Databases and Applications*, pages 177–198. Addison-Wesley Publishing Company, ACM Press, New York, USA, 1989.

E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, NY, USA, 1968.

M. Dorfman. System and Software Requirements Engineering. In H.Thayer and M.Dorfman, editors, *System and Software Requirements Engineering*, pages 4–16. IEEE Compute Society Press, Los Alamitos CA, 1990.

E. H. H. Dürr and J. vanKatwijk. VDM++: A Formal Specification Language for Object-Oriented Designs. In G. Heeg, B. Mugnusson, and B. Meyer, editors, *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 7*, pages 63–78. Prentice-Hall, 1992.

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification - 1*. Springer Verlag, Berlin, 1985.

C. A. Ellis and S. J. Gibbs. Active Objects: Realities and Possibilities. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts Databases and Applications*, pages 561–572. Addison-Wesley Publishing Company, ACM Press, New York, USA, 1989.

E. A. Emerson and J. Y. Halpern. Sometimes and not never revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1), Jan. 1986.

V. Encontre, E. Delboulbe, P. Gabaud, P. Leblanc, and A. Baussalem. Combining Services, Message Sequence Charts and SDL: Formalism Methods and Tools. Technical report, Verilog, 1990.

Excelerator. User Manual, Ver.1.2. Technical report, Index Technology Corporation, Cambridge Massachusetts, USA, 1986.

M. Felder, D. Mandrioli, and A. Morzenti. Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models. Technical report, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 91-072, Piazza Leonardo da Vinci 32, Milano, Italy, 1991.

B. Fields and M. Elvang-Goransson. A VDM Case Study in mural. *IEEE Transactions on Software Engineering*, 18(4):279–295, April 1992.

A. Forin. Real-Time, UNIX and Mach. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburg, Pa 15213, USA, 1992.

A. Gabrielian and M. K. Franklin. Multilevel Specification of Real-Time Systems. *Communications of the ACM*, 34(5):50–60, May 1991.

S. J. Garland, J. V. Guttag, and J. J. Horning. Debugging Larch Shared Language Specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1057, Sept. 1990.

GEODE. AGE/GEODE Editor, User's Manual, ver.1.4. Technical report, Verilog, avenue Artistide Briand, 52, 92220 Bagneaux, France, 1992.

R. Gerber and I. Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 68–78. IEEE Computer Society Press, Dec. 1989.

C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, Feb. 1991.

C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

H. Gomaa. A Software Design Method for Real-Time Systems. *Communications of the ACM*, 27(9):938–949, Sept. 1984.

H. Gomaa. Software Development of Real-Time System. *Communications of the ACM*, 29(7):657–668, July 1986.

H. Gomaa. A Behavioral Analysis and Modeling Method for Real-Time Systems. In *International Workshop on Real-Time Programming WRTP'92*, pages 43–48, Bruges, Belgium, 23-26 June 1992. International Federation of Automatic Control, IFAC International Federation for Information Processing, IFIP Belgian Federation of Automatic Control, IBRA-BIRA.

R. Gotzhein. Temporal logic and applications – a tutorial. *Computer Networks and ISDN Systems, North-Holland*, 24:203–218, 1992.

D. Guaspari, C. Marceau, and W. Polak. Formal Verification of ADA Programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, Sept. 1990.

J. Guttag. Abstract Data Types and Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *ACTA Informatica*, 10, 1978.

J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch Family of Specification Languages. *IEEE Software*, pages 24–36, Sept. 1985.

A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, Sept. 1990.

D. Harel. On Visual Formalism. *Communications of the ACM*, 31(5):514–530, May 1988.

D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. S.-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of Statecharts. In *Proc. 2nd IEEE Symposium on Logic in Computer Science, Ithaca, NY, USA*, pages 54–64, 22-24 June 1987.

D. J. Hatley and I. A. Pirbhai. *Strategies for Real Time System Specification*. Dorset House Publishing, New York, 1987.

C. A. R. Hoare. Towards a Theory of Parallel Programming,. In *Operating Systems Techniques*, pages 61–71. Academic Press, NY, USA, 1972.

C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

C. A. R. Hoare. A Calculus of Total Correctness for Communicating Processes. *Sci. Comput. Program.*, 1:49–72, 1981.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, NY, USA, 1985.

HOOD. An Overview of the HOOD Toolset. Technical report, Software Sciences, 1988.

Y. Ishikawa, H. Tokuda, and C. W. Mercer. An Object-Oriented Real-Time Programming Language. *Computer*, pages 66–73, Oct. 1992.

M. A. Jackson. *Principle of Program Design*. Academic Press, Inc., New York, USA, 1975.

M. A. Jackson. *System Development*. Prentice Hall International, C. A. R. Hoare Series, New York, USA, 1983.

R. J. K. Jacob. A State Transition Diagram Language for Visual Programming. *Computer*, pages 51–59, Aug. 1985.

M. S. Jaffe, N. G. Leveson, M. P. E. Heimdhal, and B. E. Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

F. Jahanian and A. K.-L. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, Sept. 1986.

S. Jahanian and D. A. Stuart. A Method for Verifying Properties of Modechart Specifications. In *Proc. of 9th IEEE Real-Time Systems Symposium*, pages 12–21, Huntsville, Ala., USA, 1988. IEEE Press.

K. Jensen. Coloured Petri nets and the Invariant-Method. *Theoret. Comput. Sci.*, 14:317–336, 1981.

K. Jensen. Coulored Petri nets. In W. Brauer, W. Resig, and G. Rozenberg, editors, *Advanced in Petri Nets 1986*. Springer Verlag, New York, USA, 1987.

C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

R. Koymans, R. K. Shyamasundar, W. P. deRoever, R. Gerth, and A. Arun-Kumar. Compositional Semantics for Real-Time Distributed Computing. In *Proc. of Logics of Programs Lecture Notes in Computer Sciences, LNCS 193*, New York, 1985. Springer Verlag.

L. Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, Jan. 1989.

L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1993.

K. Lano. Z++, An Object-Oriented Extension to Z. In J. E. Nicholls, editor, *Proc. of the 4th Annual Z User Meeting*, pages 151–172, Oxford, UK, 1991. Workshop in Computing, Springer Verlag.

G. T. Leavens and Y. Cheon. Preliminary Design of Larch/C++. In U. Martin and J. Wing, editors, *Proc. of First International Workshop on Larch*. Springer Verlag, Workshop in Computer Science Series, 1992.

N. Leveson and J. L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Transactions on Software Engineering*, 13(3):386–397, March 1987.

S.-T. Levi and A. K. Agrawala. *Real-Time System Design*. McGraw-Hill Publishing Company, New York, USA, 1990.

B. Liskov and J. Guttag. *Abstraction Specification in Program Development*. The MIT Press, Cambridge, MS, USA, 1986.

B. Liskov, A. Snyder, R. Atkinson, and G. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, Aug. 1977.

C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

LOTOSPHERE. User Manual, ESPRIT Project n.2304. Technical report, consortium LOTOSPHERE, 1992.

D. Maier and D. S. Warren. *Computing with Logic*. The Benjamin/Cummings, Inc., Menlo Park, CA, USA, 1988.

D. Mandrioli. The Specification of Real-Time Systems: a Logical Object-Oriented Approach. In *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS'92*, 1992.

D. Mandrioli. The Object-Oriented Specification of Real-Time Systems. In *Tutorial Note of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe '93*, Versailles, France, 8-11 March 1993.

D. Mandrioli, S. Morasca, and A. Morzenti. Functional test case generation for real-time systems. Technical report, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 91-072, Piazza Leonardo da Vinci 32, Milano, Italy, 1992.

M. A. Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets. In *Proc. of Performance 83, ACM Sigmetrics*, Oct. 1983.

J. Martin and J. Odell. *Object Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

G. H. Mealy. A Method for Sinthesizing Sequential Circuits. *BST Journal*, 34:1045–1079, 1955.

S. R. L. Meira and A. L. C. Cavalcanti. Modular Object Oriented Z Specifications. In J. E. Nicholls, editor, *Proc. of the 4th Annual Z User Meeting*, pages 173–192, Oxford, UK, 1991. Workshop in Computing, Springer Verlag.

P. M. Merlin and D. J. Faber. Recoverability of Communication Protocols Applications of a Theoretical Study. *IEEE Transactions on Communications*, 24, Sept. 1976.

B. Meyer. On Formalism in Specifications. *IEEE Software*, pages 6–26, Jan. 1985.

B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, C. A. R. Hoare Series, New York, USA, 1988.

R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92, Springer Verlag, New York, 1980.

J. Misra and K. M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

M. K. Molloy. Discrete Time Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 11(4):417–423, April 1985.

D. E. Monarchi and G. I. Puhr. A Research Typology for Object Oriented Analysis and Design. *Communications of the ACM*, 35(9):35–47, Sept. 1992.

A. P. Moore. The Specification and Verified Decomposition of System Requirements Using CSP. *IEEE Transactions on Software Engineering*, 16(9):932–948, Sept. 1990.

E. F. Moore. Gedanken-Experiments on Sequential Machines. In *Automata Studies, Annals of Mathematical Studies*, pages 129–153, Preston NJ, USA, 1956. Princeton Univesity Press.

T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

D. R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, 6(1):24–32, Jan 1980.

O. Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts Databases and Applications*, pages 3–22. Addison-Wesley Publishing Company, ACM Press, New York, USA, 1989.

NorthernTelecom. ObjecTime: Object-Oriented CASE for Real-Time Systems. Technical report, Bell-Northern Telecom, 1993.

E.-R. Olderog and C. A. R. Hoare. Specification-Oriented Semantics for Communicating Processes. In J. Diaz, editor, *Automata, Language and Programming, Proc. of 10th Colloquium*, pages 561–572, Barcelona, Spain, July 1983. Springer Verlag, Lecture Notes in Computer Sciences, LNCS 154.

E. R. Olderog and C. A. R. Hoare. Specification Oriented Semantics for Communicating Sequential Process. *ACTA Informatica*, 23:9–66, 1986.

J. S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press LTD., Advanced Software Development Series, 1, Taunton, Somerset, England, 1989.

J. S. Ostroff and W. Wonham. Modeling and Verifying Real-Time Embedded Computer Systems. In *Proc. IEEE Real-Time Systems Symp.*, pages 124–132. IEEE Computer Society Press, Dec. 1987.

J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Engelwood Cliffs, Prentice-Hall, NJ, 1981.

C. Petri. Kommunikation mit atomation. Technical report, Ph.D. Thesis, Schriften des Reinsh-Westfalischen Inst. Fur Instrumentelle Mathematik an der Universitat Bonn, Bonn, West Germany, 1962.

P. Pulli, R. Elmstrom, G. Leon, and J. A. delaPuente. IPTES – Incremental Prototyping Technology for Embedded real-time Systems. In *Proc. of 1991 ESPRIT Conference*, 1991.

C. Ramachandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Massachusetts Inst. Technol. Project MAC, TR. 120, USA, Feb. 1974.

C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5), Sept. 1980.

R. R. Razouk and M. M. Gorlick. A Real-Time Interval Logic for Reasoning About Execution of Real-Time Programs. In *Proc. of ACM/SIGSOFT'89 (TAV3).* ACM Press, Dec. 1989.

G. Reed and A. Roscoe. A Timed Model for Communicating Sequential Processes. In *Proc. ICALP'86*, pages 314–323. Springer Verlag, Lecture Notes in Computer Sciences, LNCS 226, 1986.

W. Reisig. *Petri Nets. An introduction.* EATCS Monographs on Theoretical Computer Science, Springer Verlag, New York, 1985.

D. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proc. of 14th International Conference on Software Engineering*, pages 105–118, Melbourne, Australia, 11-15 May 1992. IEEE press, ACM.

A. Rockstrom and R. Saracco. SDL – CCITT Specification and Description language. *IEEE Transactions on Communications*, 30(6):1310–1318, June 1982.

D. T. Ross and K. E. Schoman. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, 3(1):6–15, Jan. 1977.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, Englewood Cliffs, New Jersey, 1991.

B. Sanden. The Case for Eclectic Design of Real-Time Software. *IEEE Transaction on Software Engineering*, 35(3):360–363, March 1989a.

B. Sanden. Entity-Life Modeling and Structured Analysis in Real-Time Software Design — A Comparison. *Communications of the ACM*, 32(12):1458–1466, Dec. 1989b.

B. Sanden. An Entity-Life Modeling Approach to the Design of Concurrent Software. *Communications of the ACM*, 32(3):330–343, March 1989c.

R. Sarraco and P. A. J. Tilanus. CCITT SDL: Overview of the Language and its Applications. *Computer Networks and ISDN Systems*, 13(2):65–74, 1987.

P. A. Scheffer, A. H. StoneIII, and W. E. Rzepka. A Case Study of SREM. *Computer*, pages 47–54, April 1985.

R. L. Schwartz and P. M. Melliar-Smith. From State Machines to Temporal Logic: Specification Methods for Protocol Standards. *IEEE Transactions on Communications*, 30(12):2486–2496, Dec. 1982.

B. Selic. An Efficient Object-Oriented Variation of Statecharts Formalism for Distributed Real-Time Systems. In *Submitted to CHDL'93: IFIP Conference on Hardware Description language and Their Applications*, Ottawa, Canada, 26-28 April 1993.

B. Selic, G. Gullekson J. McGee, and I. Engelberg. ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. In *Proc. of 5th International Workshop on Computer-Aided Software Engineering, CASE'92*, Montreal, Quebec, Canada, 6-10 July 1992.

L. Sha and J. B. Goodenough. Real time Scheduling Theory and Ada. *Computer*, pages 53–62, April 1990.

A. C. Shaw. Communicating Real-Time State Machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, Sept. 1992.

S. Shlaer and S. J. Mellor. *Object Oriented Analysis: Modeling the World in Data.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1988.

S. Shlaer and S. J. Mellor. *Object Life Cycles: Modeling the World in States.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1991.

J. Sifakis. *Automatic Verification Methods for Finite State Systems, Proc. of the International Workshop Grenoble, France, June 12-14.* Springer Verlag, Lecture Notes in Computer Science, LNCS n.407, 1989.

J. M. Spivey. *The Z Notation – a Reference manual.* Prentice-Hall, New York, 1988.

J. A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, pages 10–19, Oct. 1988.

J. A. Stankovic and K. Ramamritham. *Advances in Real-Time Systems.* IEEE Computer Society Press, Washington, 1992.

STATEMATE. STATEMATE: The Languages of Statemate. Technical report, i-Logic, Inc., 22 Third Avenue, Burlington, Mass. 01803, USA, 1987.

A. D. Stoyenko. The Evolution and State-of-the-Art of Real-Time Languages. *Journal of Systems and Software*, pages 61–84, April 1992.

StP. Software through Pictures: Products and Services Overview. Technical report, Interactive Development Environment, 1991.

Bernard A. Sufrin. Formal Methods and the Design of Effective User Interfaces. In M.D. Harrison and A.F. Monk, editors, *People and Computers: Designing for Usability.* Cambridge University Press, UK, 1986.

C. A. Sunshine, D. H. Thompson, R. W. Erickson, S. L. Gerhart, and D. Schwabe. Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models. *IEEE Transactions on Software Engineering*, 8(5):460–489, Sept. 1982.

C. P. Svoboda. Structured Analysis. In H.Thayer and M.Dorfman, editors, *System and Software Requirements Engineering*, pages 218–237. IEEE Compute Society Press, Los Alamitos CA, 1990.

B. Taylor. A Method for Expressing the Functional Requirements of Real-Time Systems. In *Proc. of 9th IFAC/IFIP Conference of Real-Time Programming*, pages 111–120. New-York: Pergamon, 1980.

Teamwork. User Manuals, Ver. 4.0. Technical report, Teamwork Division of CADRE, Providence, R.I., USA, 1992.

H. Thayer and M. Dorfman. *System and Software Requirements Engineering.* IEEE Compute Society Press, Los Alamitos CA, 1990.

H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proc. Usenix Mach Workshop*, pages 1–10, Oct. 1990.

I. J. Walker. Requirements of an Object-Oriented Design Method. *Software Engineering Journal*, pages 102–113, March 1992.

P. T. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. *IEEE Transactions on Software Engineering*, 12(2):198–210, Feb. 1986.

P. T. Ward and S. J. Mellor. *Structured Development for Real-time Systems.* Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

R. R. Willson and B. H. Krogh. Petri Net Tool for the Specification and Analysis of Discrete Controllers. *IEEE Transactions on Software Engineering*, 16(1):39–50, Jan. 1990.

J. M. Wing. Writing Larch Interface language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, Jan. 1987.

J. M. Wing. A Specifier's Introduction for Formal Methods. *Computer*, pages 8–24, Sept. 1990a.

J. M. Wing. Using Larch to Specify Avalon/C++ Objects. *IEEE Transactions on Software Engineering*, 16(9):1076–1088, Sept. 1990b.

R. J. Wirfs-Brock, B. Wilkerson, and L. Winer. *Designing Object Oriented Software.* Prentice Hall, Englewood Cliffs, N.J., USA, 1990.

R. Wirsf-Brock and B. Wilkerson. Object-Oriented Design: a responsibility-driven approach. In *Proc. OOPSLA'89*, pages 71–75, New Orleans, Louisiana,, Oct. 1989. SIGPLAN NOT, ACM Press.

E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1979.

P. Zave. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, May 1982.

P. Zave. The Operational Versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):104–118, Feb. 1984.

P. Zave. A Comparison of the Mayor Approaches to Software Specification and Design. In H.Thayer and M.Dorfman, editors, *System and Software Requirements Engineering*, pages 197–199. IEEE Compute Society Press, Los Alamitos CA, 1990.

P. Zave and W. Schell. Salient Features of an Executable Specification Language and Its Environment. *IEEE Transactions on Software Engineering*, 12(2):312–325, Feb. 1986.