# The Specification of Real-Time Systems: a Logical, Object-Oriented Approach

**Dino Mandrioli**
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy.
Presently on leave at the Department of Computer Science, University of California at Santa Barbara.

## Abstract

Several requirements for high quality specifications of real-time systems are stated. An approach based on the integration of logic and object-oriented formalisms is advocated and the essential features of a language–named TRIO$^+$–that exploits such an approach are described. It is also argued that the benefits of the use of rigorous approaches to the system specification phase can be highly enhanced by suitable CASE tools that support the whole process. Early experiences on the application of the TRIO$^+$ language and of its tools to industrial problems are reported.

## 1. Introduction

There is a general consensus that the early phases in the design of any system are the most critical and error prone. This fact is even more striking in the case of software design, where, for a long time, early phases have not been supported by adequate methods and tools. They are receiving, however, growing attention in these days.

A key part of early design phases are *specifications*, which span from *requirement*, to *functional*, to *design, specifications*. More generally, by the term "specification" we mean the *definition* of something that must be *implemented* and *verified* against the given definition [GJM].

Specifications serve to different purposes:

- They can be the basis of a contract between the user and the producer of the system.
- They are the reference to drive design decisions which must achieve the specified requirements.
- They are the reference to verify the correctness of the realized system.
- They are the reference during the maintenance phase: if such a maintenance is corrective, previously unmatched requirements must be met by the new implementation; if it is perfective, the consistency with the requirements must be maintained; if it is adaptive, most often new requirements are stated so that modifications start with specifications themselves.

As a consequence of the above purposes, specifications should satisfy the following requirements, which are, sometimes, in conflict each other.

- They should be *clear*, *precise*, and *unambiguous*. Experience shows that subtle ambiguities are often hidden in specification documents and that this is the cause of major inconveniences and even of disasters. The use of natural languages is often considered a major source of this problem.
- They should be easily *understandable*. The consequences of this requirement are different, depending on who is in charge of understanding them properly, whether he/she is an end-user with no technical background or a designer, expert in the use of technical documentation and formalisms. Sometimes this requirement *seems* in conflict with the previous one.
- They should be *natural*. I.e., they should be expressed in a way that easily fits with the given application. This may have the consequence that different specification styles/techniques/ languages could be used in different contexts–say, in office automation systems or in embedded applications.

- They should not bias towards specific design decisions. This requirement is often referred as the "what versus how" paradigm. Although in many practical cases the distinction is not so sharp (often the best way to explain what one wants is to give an example thereof), the principle should always be kept in mind to avoid overspecification.

Further requirements are more or less consequences of the previous ones.

- Since it is quite likely that specifications of complex systems are complex, and complex things are rarely right in their first formulation:
  - Specifications should be *well structured* and *modularized.*
  - Specifications should be obtained *incrementally* by stepwise refining several versions thereof starting from initial, imprecise, and incomplete versions to finally achieve precise and complete formulations. In this process, several *specification languages*, spanning from natural language, to drawings, to mathematical formalisms, etc. could be used.
- As well as any other product of human activity, specification quality may highly benefit from the use of appropriate *methods* and *tools*. Such tools should help their:
  - editing;
  - analysis (tools that help specification analysis include consistency verifiers, simulators or prototypers, ...);
  - managing;
  - integration with the products of other design phases (tools to drive the implementation from the specifications, tools to help verification, such as test case generators, ...)

The above requirements are even more critical in the application field of *real-time systems*, since most often such systems have strongest reliability requirements (safety-critical systems) such as in the case of plant control systems, embedded applications, medical systems, etc.

Real-time systems also impose the ability to deal explicitly with the time variable, since their correctness does depend on the relative speed of the involved processes and their requirements include timeliness[1].

The state of the art of software specification exhibits a fairly rich set of methods, languages and tools, although a clear understanding of relative pros and cons is still needed and their systematic application in the industrial world still leaves much to be desired.

Rather than trying to list such methods and related tools, what would result necessarily into a long, tedious, and incomplete sequence, we believe it is worth proposing the following classifications thereof (in the following we will refer to specification methods, languages, and tools, as *specification frameworks*.)

- Specification frameworks can be classified according to their *level of formality*. Here we distinguish among:
  - *Informal frameworks*. They include natural languages and figures of any type. They are natural and easy to use but expose to the risk of ambiguities. The only automatic support they can receive is from editing and configuration tools. Nevertheless, they are still far the most widely used ones.
  - *Formal frameworks*. They are based on mathematical models and have a precisely defined syntax and semantics. They have dual advantages and disadvantages w.r.t. informal frameworks: they avoid ambiguities but are often criticized–mainly in the industrial world–because they are difficult to learn and to use. In our opinion, however, most of their disadvantages may be overtaken by applying common sense and incrementality (specifications should be incremental even w.r.t. the level of formality; furthermore, often, the use of formalism may be restricted to the

[1]This is in sharp contrast with traditional systems, such as transaction handlers, where time requirements do exist but are a matter of performance with no impact on functional correctness: if alarm recovery is performed out of the deadline it has the same effect as no recovery at all, i.e., the disaster; whereas a late response in an automatic teller does not affect the correctness of the checking account of the customer (only his patience is affected!).

specification of few critical cases, not generalized to a whole system), and by the use of well designed support tools. This issue will be further clarified later since it is a major topic of the present paper.

- *Semiformal frameworks*. By this term we mean notations for which a more or less precise syntax is defined but whose meaning is left to the user intuition. A typical example of such a semiformalism are the classical Data-Flow-Diagrams. Semiformal frameworks have obviously intermediate features between informal and formal ones.

- Specification frameworks can be categorized according to the kind of user interface they exploit. In this respect, it is quite natural to see a movement from original textual interfaces towards graphical and integrated notations, with typical hypertext features.
- Specification frameworks can also be classified on the basis of the nature of their underlying (mathematical) models. Here we distinguish between:
  - *Operational frameworks*. They are based on abstract machines that in some way simulate the system under specification. The simplest and most widely known example of such a model are Finite State Machines. Petri nets [Pet] are more and more adopted for the description of concurrent, possibly real-time, systems.
  - *Descriptive frameworks*. They aim at specifying the desired system properties, without giving a model of the system structure, not even at the very high level of an abstract machine. Algebra and mathematical logic provide a natural basis for many of such frameworks.

  Not surprisingly, operational and descriptive models have symmetric properties too.

  In general, operational models are easier to learn, to understand–at least for users who are not specialized in mathematics–, and to simulate. On the other hand they tend to bias towards implementation decisions and require another language to express their properties. For instance, the fact that a Petri net is live cannot be expressed as a Petri net; on the contrary, once the behavior of a plant has been formalized through a set of logical formulas, the same logic language can be used to express the fact that the plant will never crash and the fact can be proved as a *theorem* of the language.

  After much debate on the relative weights of pros and cons of the two approaches, some agreement is now arising that the two are actually nicely complementary and should be used in conjunction rather than in mutual exclusion. For instance, [FMM], [Ost] advocate the use of logic languages to state properties of systems that are modeled as (extended) Petri nets or as (extended) Finite State Machines, respectively.

In most cases, the specification frameworks that are in use in industrial environments are either completely informal of semiformal; they are operational in nature; and sometimes they are supported by tools which are well engineered from the point of view of user interface and configuration management.

Most of the tools that are presently commercially available are based on the classic–but also old-fashioned and criticized–Structured Analysis methodology. This is also due to the natural delay between the development of new models and methodologies and the construction of well-engineered tools.

Noticeable exceptions to the above picture are frameworks based on formal operational models such as Finite State Machines (e.g., Satecharts [Har et al.]), or Petri nets (e.g., Artiflex, RdP). VDM [BjP] and Z [Spi] are two formal specification languages, fairly well adopted in the industrial world, whose theoretical background is descriptive in nature (denotational semantics) although their practical use is actually a mixture of descriptive and operational approaches. They do not seem well-suited, however, to deal deeply with concurrent, and, mostly, with real-time, systems.

Object-Oriented analysis techniques (in general rather operational in nature) are presently advocated by many researchers, mainly as an alternative to the old-fashioned function-oriented approach typical of structured analysis [ShM],

[CHB]. In fact, it turns out that most of the reasons supporting the object-oriented approach to software design (modularity and abstraction to master complex systems, reusability, incrementality, parameterizabilty) still apply to the specification phase. For instance, we may wish *specification libraries* containing the specification of fairly standard components in some application field, to be specialized and integrated with ad-hoc specifications of new components to obtain a final specification of a whole system.

This paper briefly presents TRIO$^+$, a specification language for real-time systems which has been developed at Politecnico di Milano. It is the kernel of a complete specification and design environment which is under development.

TRIO$^+$ aims at achieving most of the above requirements for high quality-specifications of real-time systems through the following major features.

- It exploits mathematical logic to describe in a fully rigorous way system requirements.
- It deals explicitly with the time variable to address specifically real-time issues.
- It exploits object-oriented features to obtain well-structured, modular specifications and to achieve incrementality at any level.
- It is supported by a collection of tools, some of which are still under development, that allow not only the editing but also the semantic analysis of the specifications, thanks to the fact that the language is executable.

The paper is structured as follows.

Section 2 briefly summarizes TRIO, which is a simple extension of traditional temporal logic to deal explicitly with time measures[1]. TRIO$^+$ is then presented in Section 3 as a high level language, based on TRIO, that is suitable to deal with the specification of large, real life, systems. In two summarizing "equations", the relations between the two languages could be explained as follows:

1. TRIO/TRIO$^+$ =
   Algol-like language of toy type/Pascal or Ada;
2. TRIO$^+$ =
   TRIO (i.e., metric temporal logic) +
   object-oriented mechanisms.

Section 4 describes the whole environment centered around TRIO, its present state of development, and its projected evolution. Finally, Section 5 reports on the experience gained so far in the application of TRIO$^+$ to industrial projects.

Since the purpose of this paper is of tutorial type, we will just try to give the flavor of language's essential features, mainly through simple examples. The reader interested in more technical details is referred to the more specialized reports mentioned in the bibliography.

## 2. TRIO: an extension of temporal logic

TRIO is a first order logical language, augmented with temporal operators which permit to talk about the truth and falsity of propositions at time instants different from the current one, which is left implicit in the formula. We now briefly sketch the syntax of TRIO and give an informal and intuitive account of its semantics; detailed and formal definitions can be found in [GMM].

Like in most first order languages, the alphabet of TRIO is composed of variable, function, and predicate names, plus the usual primitive propositional connectors '$\neg$' and '$\rightarrow$', the derived ones '$\wedge$', '$\vee$', '$\leftrightarrow$', ..., and the quantifiers '$\exists$' and '$\forall$'.

In order to permit the representation of change in time, variables, functions, and predicates are divided into *time dependent* and *time independent* ones. Time dependent variables represent physical quantities or configurations that are subject to change in time, and time independent ones represent values unrelated with time. Time dependent functions and predicates denote relations, properties, or events that may or may not hold at a given time instant, while time independent functions and predicates represent

[1] We recall that, in spite of its name, traditional temporal logic can only express requirements of the type "event B will *eventually* occur as a consequence of event A", without specifying explicitly *when*–i.e., within which deadline–B will occur.

facts and properties which can be assumed not to change with time.

TRIO is a typed language, since we associate a domain of legal values to each variable, a domain/range pair to every function, and a domain to all arguments of every predicate. Among variable domains there is a distinguished one, called the *Temporal Domain*, which is numerical in nature: it can be the set of integer, rational, or real numbers. Functions representing the usual arithmetic operations, like '+' and '-', and time independent predicates for the common relational operators, like '=', '≠', '<', '≤', are assumed to be predefined at least for values in the temporal domain.

TRIO formulas are constructed in the classical inductive way. A *term* is defined as a variable, or a function applied to a suitable number of terms of the correct type; an atomic formula is a predicate applied to terms of the proper type. Besides the usual propositional operators and the quantifiers, one may compose TRIO formulas by using primitive and derived *temporal operators*. There are two temporal operators, *Futr* and *Past*, which allow the specifier to refer, respectively, to events occurring in the future or in the past with respect to to the current, implicit time instant. They can be applied to both terms and formulas, as shown in the following. If *s* is any TRIO term and *t* is a term of the temporal type, then

Futr (s, t) and Past (s, t)

are also TRIO terms. The intended meaning is that the value of *Futr(s, t)* (resp. *Past(s, t)*) is the value of term *s* at an instant laying *t* time units in the future (resp. in the past) with respect to the current time instant. Similarly, if *A* is a TRIO formula and *t* is a term of the temporal type, then

Futr (A, t) and Past (A, t)

are TRIO formulas too, that are satisfied at the current time instant if and only if property *A* holds at the instant laying *t* time units ahead (resp. behind) the current one.

On the basis of the primitive temporal operators *Futr* and *Past*, numerous derived operators can be defined for formulas. We mention, among the many possible ones, the following:

$$\text{AlwF}(A) \stackrel{\text{def}}{=} \forall t\,(t > 0 \rightarrow \text{Futr}(A, t))$$

$$\text{AlwP}(A) \stackrel{\text{def}}{=} \forall t\,(t > 0 \rightarrow \text{Past}(A, t))$$

$$\text{SomF}(A) \stackrel{\text{def}}{=} \neg\,\text{AlwF}(\neg A)$$

$$\text{SomP}(A) \stackrel{\text{def}}{=} \neg\,\text{AlwP}(\neg A)$$

$$\text{Som}(A) \stackrel{\text{def}}{=} \text{SomP}(A) \vee A \vee \text{SomF}(A)$$

$$\text{Alw}(A) \stackrel{\text{def}}{=} \text{AlwP}(A) \wedge A \wedge \text{AlwF}(A)$$

$$\text{Lasts}(A, t) \stackrel{\text{def}}{=} \forall t'\,(0 < t' < t \rightarrow \text{Futr}(A, t'))$$

$$\text{Lasted}(A, t) \stackrel{\text{def}}{=} \forall t'\,(0 < t' < t \rightarrow \text{Past}(A, t'))$$

$$\text{Until}(A_1, A_2) \stackrel{\text{def}}{=} \exists t\,(t > 0 \wedge \text{Futr}(A_2, t) \wedge \text{Lasts}(A_1, t))$$

$$\text{Since}(A_1, A_2) \stackrel{\text{def}}{=} \exists t\,(t > 0 \wedge \text{Past}(A_2, t) \wedge \text{Lasted}(A_1, t))$$

$$\text{NextTime}(A, t) \stackrel{\text{def}}{=} \text{Futr}(A, t) \wedge \text{Lasts}(\neg A, t)$$

$$\text{LastTime}(A, t) \stackrel{\text{def}}{=} \text{Past}(A, t) \wedge \text{Lasted}(\neg A, t)$$

The intuitive meaning of these derived temporal operators stems from that of the basic *Futr* and *Past* temporal operators, according to the kind of quantification they contain. Thus, AlwF(A) means that A will hold in all future time instants; SomF(A) means that A will hold sometimes in the future; Lasts(A, t) means that A will hold for the next t time units; Until($A_1$, $A_2$) means that $A_2$ will happen in the future and $A_1$ will be true until then; NextTime (A, t) means that the first time in the future when A will hold is t time units apart from now. The meaning of the operators AlwP, SomP, Lasted, Since, and LastTime is exactly symmetrical to that of the corresponding operators regarding the future. Som(A) means that there is a time instant–in the past, now, or in the future– where A holds. Alw(A) means that A holds in every time instant of the temporal domain.

The reader can easily find in the above list of TRIO derived temporal operators the operators of classical temporal logic. For instance, SomF corresponds the “Eventually” operator of temporal logic. Thus, TRIO is an extension of temporal logic as it allows not only to describe a temporal ordering in the occurrence of events, but also to quantify their distance in the temporal domain.

**Example 2.1**

As a first example of TRIO formulas, consider a pondage power station, where the quantity of water held in the reservoir is controlled by means of a sluice gate. The gate is controlled via two commands, *up* and *down*, which, respectively, open and close it, and are represented as a TRIO time dependent predicate named *go* with an argument in the range {*up*, *down*}. The current state of the gate can have one of the four values: *up* and *down* (with the obvious meaning), and *mvUp*, *mvDown* (meaning respectively that the gate is being opened or closed). The state of the gate is modelled in TRIO by a time dependent variable, called *position*. The following formula describes the fact that it takes the sluice gate $\Delta$ time units to go from the *down* to the *up* position, after receiving a go(up) command.

$$\left(\begin{array}{c}\text{position} = \text{down} \\ \wedge \\ \text{go(up)}\end{array}\right) \rightarrow \left(\begin{array}{c}\text{Lasts}(\text{position} = \text{mvUp}, \Delta) \\ \wedge \\ \text{Futr}(\text{position}=\text{up}, \Delta)\end{array}\right)$$

When a *go(up)* command arrives while the gate is not still in the *down* position, but is moving down because of a preceding *go(down)* command, then the direction of motion of the gate is not reversed immediately, but the downward movement proceeds until the *down* position has been reached. Only then the gate will start opening according to the received command.

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

If the behavior of the sluice gate is symmetrical with respect to its direction of motion, two similar TRIO formulas will describe the commands and their effects in the opposite direction. $\rightarrow$

## 2.1. TRIO's formal semantics and executability

As mentioned above, a major feature of the TRIO language is its *executability*. This allows the construction of semantic tools that help specification validation and implementation verification, e.g., through prototyping (specification simulation) and through test case generation, respectively (see Section 4 for a few hints on TRIO's tools).

Executability is achieved in TRIO along the following lines (details can be found in [GMM], [MMG]).

A model-theoretic semantics is defined for the language, i.e., an interpretation schema is given that, for any TRIO formula, aim at building possible *models* of the formula[1].

In general, since TRIO includes first-order arithmetics, the satisfiability of arbitrary formulas (i.e., stating whether or not there exists a model for a formula) is undecidable. Thus, the general interpretation algorithm mentioned above is not guaranteed to terminate with a definite answer.

Partial executability is however obtained by exploiting the idea of *finite approximation of infinite domains*. Original interpretation domains, which are usually infinite, are replaced by finite approximations thereof. For instance, the set of integers is replaced by the interval [0..1000]. By this way, every decision problem becomes decidable. Of course, there is no a priori warranty that the result obtained in the finite domain coincides with the theoretical result that would be obtained on the infinite domain. In practice, however, we may often rely on this type of *specification testing* [Kem] on the basis of the following considerations.

- Quite often our common sense and experience can tell us that if the system we are specifying and implementing has a dynamic behavior with a time evolution of the order of magnitude of, say, the seconds, then, after having tested it for several hours all "relevant facts" about it have been generated.
- We may try several executions with different time domains of increasing cardinality (domain

[1] We recall that a model of a logic formula is an assignment of values to variables and predicates occurring in the formula, such that the formula evaluates to TRUE. In the case of TRIO, an assignment to a time-dependent variable or predicate means assigning a value in the suitable domain for each instant in the time domain. For this reason a model of a TRIO formula is also called a *history* thereof.

$T_{i+1}$ strictly includes domain $T_i$). If we detect that, from some $T_k$, the result of formula interpretation does not change, we may infer that it will not change forever in the future. In a similar way, several numerical algorithms are considered as sufficiently "stable" if their results do not change too much from one iteration to another one, even if no mathematical proof of convergence exists.

Even with the use of finite approximations of infinite domains, general interpretation algorithms are often intractable from the point of view of computational complexity.

Nevertheless, they did produce results of practical impact (see Section 5) thanks to the following facts:

- Well written TRIO specifications are in general a wide collection of small, largely independent, formulas that can be tested separately, rather than huge, single, formulas. Here, the modularization techniques that will be introduced in the next section play a central role.
- Specialized and simplified versions of the general algorithms have been built, that exploit several particular cases.

## 3. TRIO$^+$: an object-oriented language based on TRIO

TRIO has proved to be a useful specification tool, since it combines the rigor and precision of formal methods with the expressiveness and naturalness of first order and modal logics. The use of TRIO for the specification of large and complex systems, however, has shown its major flaw: as originally defined, the language does not support directly and effectively the activity of structuring a large and complex specification into a set of smaller modules, each one corresponding to a well identified, autonomous subpart of the system that is being specified. This is because TRIO specifications are very finely structured: the language does not provide powerful abstraction and classification mechanisms, and lacks an intuitive and expressive graphic notation. In summary, TRIO is best suited to the specification "in the small", that is, to the description of relatively simple systems via formulas of the length of a few lines.

In the description of large and complex systems [CHJ], however, one often needs to structure the specification into modular, independent, and reusable parts. In such a case, beyond formality, executability, rigor, and absence of ambiguity, other language features become important, such as the ability to structure the specifications into modules, to define naming scopes, to produce specifications by an incremental, top-down process, to attribute a separate level of formality and detail to each portion of the specification [MBM]. These issues are similar to those arising in the production of large programs, an activity that is usually called "programming-in-the-large" [DeRK]. Hence, we may refer to the process of producing specifications of complex systems as *specifying-in-the-large*.

To support specification in the large, TRIO has been enriched by concepts and constructs from object-oriented methodology, yielding a language called TRIO$^+$. Among the most important features of TRIO$^+$ are the ability to partition the universe of objects into classes, inheritance relations among classes, and mechanisms such as genericity to support the reuse of specification modules and their top-down, incremental development. Structuring the specification into modules supports an incremental, top-down approach to the specification activity through successive refinements, but also allows one to build independent and reusable subsystem specifications, that could be composed in a systematic way in different contexts. Also desirable is the possibility of describing the specified system at different levels of abstraction, and of focusing with greater attention and detail on some more relevant aspects, leaving unspecified, or less formalized, other parts that are considered less important or that are already well understood.

TRIO$^+$ is also endowed with an expressive graphic representation of classes in terms of boxes, arrows, and connections to depict class instances and their components, information exchanges, and logical equivalences among (parts of) objects. In principle, the use of a graphic

notation for the representation of formal specifications does not improve the expressiveness of the language, since it provides just an alternative syntax for some language constructs. In practice, however, the ability to visualize constructs of the language and use their graphic representation to construct, update or browse specifications can make a great difference in the productivity of the specification process and in the final quality of the resulting product, especially when the graphic view is consistently supported by means of suitable tools, such as structure-directed editors, consistency checkers, and report generators.

The following of this section summarizes the main features of TRIO$^+$ and is organized as follows.

Section 3.1 introduces TRIO$^+$ classes, which are either simple (Section 3.1.1) or structured (Section 3.1.2). Sections 3.2 and 3.3 introduce genericity and inheritance in TRIO$^+$, respectively.

## 3.1. TRIO$^+$ classes

TRIO$^+$ classes denote collections of objects that satisfy a set of axioms. As stated above, they can be either simple or structured–the latter term denoting classes obtained by composing in some way simpler ones: the two categories are introduced in the next subsections.

### 3.1.1. Simple classes

Essentially, a simple class is defined through a set of TRIO axioms premised by a declaration of all items that are referred therein in much the same way as traditional Pascal-like programs consist of a declarative part followed by an executable part.

The main syntactic features of such classes are explained through the following example.

**Example 3.1**

The following definition restates in TRIO$^+$ the sluice gate specification of Example 2.1.

**class** sluice_gate

**visible** go, position

**temporal domain** integer

**TD Items**

**predicates** go({up, down})

**vars** position: {up, down, mvup, mvdown}

**TI Items**

**consts** $\Delta$: integer

**axioms**

**vars** t: integer

*go_down:* position=up $\wedge$ go(down ) $\rightarrow$
Lasts (position=mvdown, $\Delta$) $\wedge$
Futr (position=down , $\Delta$)

*go_up:* position=down $\wedge$ go(up) $\rightarrow$
Lasts (position=mvup, $\Delta$) $\wedge$
Futr (position=up, $\Delta$)

*move_up:* position=mvup $\wedge$ go(down ) $\rightarrow$
$\exists$t NextTime(position=up,t) $\wedge$
Futr(Lasts (position=mvdown, $\Delta$) $\wedge$
Futr(position=down,$\Delta$), t)

*move_down*: position=mvdown $\wedge$ go(up) $\rightarrow$
$\exists$t NextTime (position=down, t) $\wedge$
Futr(Lasts (position=mvup, $\Delta$) $\wedge$
Futr(position=up, $\Delta$), t)

*reliability:* -- the Mean Time to Time Failure of the engine must be greater than 5 years --

**end** sluice_gate $\rightarrow$

The example suggests a natural partition of the class definition into a header, a declaration of several elements, and a set of axioms.

The class *header*, which gives the name of the class, is followed by the **visible** clause, which defines the class *interface*. In the example, *go* and *position* are the only available symbols when referring to modules of the class sluice_gate in the axioms of another class. The **temporal domain** clause determines the temporal domain of the specification. The domain is always numerical in nature: for instance, it can be the set of integer, real, or rational numbers. Here, discrete time is

considered. The keyword **TD Items** is followed by the declarations of the local time dependent functions, predicates, and variables; the keyword **TI Items** is followed by the local time independent functions, predicates, and constants. The declarations are based on predefined scalar types, such as integer, real, boolean, finite sets, subranges. In the example $\Delta$ is an integer constant, *go* is a unary time dependent predicate on the set {up, down}, *position* is a time dependent variable whose values may range on {up, down, mvup, mvdown}.

The **axioms** are TRIO formulas or natural language sentences. This is to allow the description of a system that mixes formal and informal specifications. TRIO$^+$ specifications can be completely formal, but this is not strictly mandatory. It is even possible to specify a system in natural language only, using TRIO$^+$ just to structure the specification. These are two extreme situations: in practical cases, intermediate degrees of formality can be successfully used.

The TRIO formulas of the **axioms** are prefaced with an implicit universal temporal quantification, i.e. an *Always* temporal operator. For instance, the first axiom in the sluice_gate class is to be understood as:

$$\text{Always}\big(\text{position} = \text{down} \wedge \text{go (up)} \rightarrow \text{Lasts (position} = \text{mvup}, \Delta) \wedge \text{Futr (position} = \text{up}, \Delta)\big)$$

A name can precede an axiom, to be used as a reference for axiom redefinition in inheritance (see Section 3.3). The name must be different from the names of the items of the class. A **vars** clause between the keyword **axioms** and the axioms can be used to declare the time independent variables occurring in the axioms. The items and the variables of a class (including the inherited ones) are the only symbols of variables, predicates, and functions which can occur in class axioms. This rule will be extended in Section 3.3.

#### *3.1.1.1. Class instances in TRIO$^+$*

In classical object-oriented languages of procedural type such as Eiffel, Smalltalk, C++, the notion of *object as class instance* is in some sense an abstraction of a memory cell, i.e., an information container whose contents obeys the class laws.

On the contrary, TRIO$^+$ is a pure logic language. Thus, a TRIO$^+$ variable must be intended in a purely mathematical sense, not in the traditional sense of procedural language.

As a consequence, an *instance* of a class is a *model* for the axioms of the class, i.e., an interpretation for all entities declared in the **Items** clause, such that all the axioms are true. So, a class declaration is the intensional representation of the set of its models. For instance, the following table is a partial representation of a class instance of sluice_gate (for the sake of brevity only four instants of the temporal domain are considered here):

| *field name* | *value* |
|---|---|
| **position** | ⟨ down, mvup, mvup, up, ...⟩ |
| **go** | ⟨ {up}, {} ,{} ,{}, ...⟩ |
| **Δ** | 3 |

**Table 3.1**. An instance of the sluice_gate class specified in Example 3.1

Intuitively, the time dependent **Items** of an instance represent one complete possible evolution of the specified system. The value for *go* is a sequence of unary relations, one for every instant of the temporal domain. In the example, go(up) is true in the first instant, and from the following moment the sluice_gate is moving (position=mvup) and go(.) is false (empty relation); $\Delta$ time units after the command, the gate is up (position=up).

Therefore, the fact that TRIO$^+$ has *no* primitives like *Create* or *New* to explicitly control instance creation should generate no surprise.

#### *3.1.1.2. The graphical representation of TRIO$^+$ classes*

A class may have a meaningful graphic representation as a box, with its name written at the left top; the name of the items are written on lines internal to the box; if an item is visible, then the corresponding line continues outside the box.

The previous class *sluice_gate* is graphically represented in Figure 3.1.

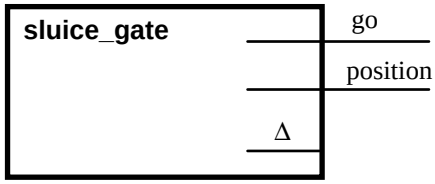


**Figure 3.1.** Graphic representation of the class sluice_gate of Example 3.1

Presently, the **axioms** part of a TRIO$^+$ class definition is only textual.

### 3.1.2. Structured classes

The fundamental technique for programming in the large is modularization, using the information hiding principle. In object-oriented languages modularization is obtained by declaring *classes*, that describe sets of objects. A class may have components of other classes, i.e., every instance of the class may contain parts that are instances of those classes. For example, a reservoir may contain two sluice gates, one for water input and the other for output: an object-oriented description of this simple system consists of a class which has two components of type sluice_gate, representing distinct and separately evolving objects. Classes which have components–called modules–of other classes are called *structured classes*. They permit the construction of TRIO$^+$ modular specifications, especially suited to describe systems in which *parts*, i.e., *modules*, can be easily recognized. A first illustration of the notion of structured class is provided by the following example.

**Example 3.2**

The above mentioned reservoir may be defined as having two modules of the class sluice_gate. This is achieved by the following class declaration (for the sake of simplicity, the example has no items and no axioms):

**class** reservoir -- first draft version --

**temporal domain** integer

**visible** inputGate.go, outputGate.go

**modules** inputGate, outputGate: sluice_gate

**end** reservoir

Recursive definitions of classes are not allowed: so a class (and its subclasses: see Section 3.3) cannot be used to declare its own modules. The temporal domain must be the same for a class and for its modules. Structured classes have an intuitive graphic representation: the modules of the class are just boxes, with a name and a line for every visible item. The picture for the reservoir example is given in Figure 3.2.a.

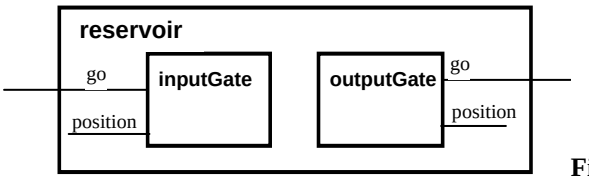


**Figure 3.2.a.** The graphic representation of class reservoir (draft version)

Modules cannot be used directly in axioms, because they are not logical symbols such as predicate or function names: they represent a set of item and module definitions, with related axioms. For the same reason the visible interface cannot list entire modules, but only their visible items, such as *inputGate.go*. The visible items of a module can be accessed in the axioms of an enclosing class by using a dot notation. For example, the following is a possible axiom of the class *reservoir*, stating that outputGate cannot be up when inputGate is down:

$$\text{inputGate.position} = \text{down} \rightarrow \text{outputGate.position} \neq \text{up}.$$

Instead, an axiom containing *outputGate.area* would be incorrect, because *area* is not a visible item of the class *sluice_gate*. →

#### *3.1.2.1. Connections between classes*

TRIO$^+$ supports some more facilities to specify complex real world systems. One facility tries to extend the expressiveness of the graphic notation. We illustrate it by enlarging the previous example.

**Example 3.3**

A more realistic reservoir than that of Example 3.2 may have two actuators, one to control each sluice gate, and a transducer which measures the level of the reservoir. The external plant is able to send four commands to control the reservoir, to open or close each sluice gate. This can be

described by defining the following class *reservoir*, whose graphical form is depicted in Figure 3.2.b.

**class** reservoir

**visible** transducer**.**level, openInput, closeInput, openOutput, closeOutput

**temporal domain** integer

**TD Items**

**predicates** openInput, closeInput, openOutput, closeOutput

**modules** inputGate, outputGate: sluice_gate

transducer: cl_transducer

actuator1, actuator2: actuator

**connections**

{ (openInput actuator1.open)

(closeInput actuator1.close)

(openOutput actuator2.open)

(closeOutput actuator2.close)

(actuator1.go inputGate.go)

(actuator1.position inputGate.position)

(actuator2.go outputGate.go)

(actuator2.position outputGate.position)

}

**axioms**

*Ax1:* inputGate.position = down → outputGate.position ≠ up

**end** reservoir

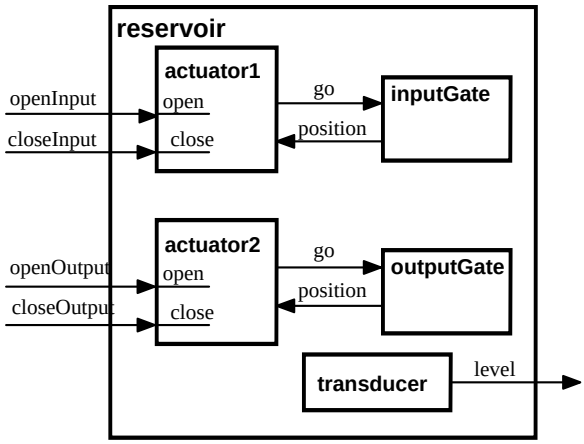


**Figure 3.2.b.** The graphic representation of the class *reservoir* (final version).

Every instance of this class contains two instances of sluice_gate, one instance of *cl_transducer* and two instances of *cl_actuator*. We assume classes *cl_transducer* and *cl_actuator* as already defined. The interface of *cl_transducer* includes a time dependent variable *level*, and the interface of *cl_actuator* contains the propositional time dependent variables *open* and *close*, plus *go* and *position* with the same meaning as in *sluice_gate*.

**Connections** is a list of pairs, denoting equivalence or identity between two items in the current scope. A connection is pictorially represented by a line joining the two items. If the two items have the same name, then this is repeated only once, near the linking line. Connections can often be interpreted as information flows between parts; it is then possible to use an arrow to represent the direction of the flow. However, the direction of arrows has no associated semantics, since there is no real distinction between the two connected items: it is only an informal, although expressive, notation.

In the example, the commands openInput, closeInput, openOutput, closeOutput and the instantaneous value of reservoir level are all that the external world is allowed to know of a reservoir. The connections state that the commands are not sent directly to the gates, but to the actuators, which control the gates and decide when moving them up and down. →

Connections are a useful tool for the specification of a complex system: the user describes system components separately; then he/she identifies

information flows between them. A structured class can be thought of as a complex system, composed of interacting subsystems; interactions are shown by the connections between modules.

*3.1.2.2. Array of classes*

Another TRIO$^+$ facility is a construct to describe those systems which may contain groups of identical parts: for instance, a shift register is composed of a certain number of DT flip-flop's, a power generation station may have a group of generators working in parallel, a reservoir may contain a set of sluice gates, and so on. These situations are easily described in TRIO$^+$, thanks to the use of *arrays* of modules. For example, a class specifying a four-gate reservoir may be built as follows:

```
class 4_gate_reservoir
  ....
  modules gate: array [1..4] of sluice_gate
  -- an array of four modules,
        accessed as gate[.] --
  ....
end 4_gate_reservoir
```

As well as in traditional Pascal-like languages, the user can access elements of an array by indexing. So, *gate[2].position* is the component *position* of the 2nd element of the array *gate*. The graphical representation of array uses multiple borders, regardless of the dimension of the array:, as suggested by Figure 3.3.

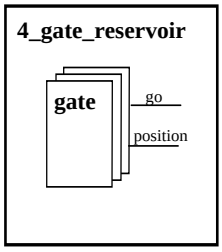


**Figure 3.3.** Graphical display of arrays of classes

Enumeration types, for instance (input, output, emergencyInput, emergencyOutput), can be used as array ranges.

## 3.2 Genericity

TRIO$^+$ is provided with a genericity mechanism (see [Mey86] for a good introduction to genericity in object oriented languages). Generic classes have one or more parameters. Parameters can be types or classes. A simple use of genericity is explained through the following example.

**Example 3.4**

Consider the specification of a reservoir containing an array of sluice gates working in parallel, say to minimize the effect of a gate failure. The number of gates can vary with the particular plant, but the structure of the plant is unchanged from one plant to another. A TRIO$^+$ specification of such a system can be achieved with the following generic class:

```
class multi_gate_reservoir [GateRange]
        -- GateRange is a parameter --
    ....
    modules gate: array [GateRange] of
                  sluice_gate
    ....
end multi_gate_reservoir
```

The specification of a particular reservoir is obtained by instantiating the formal parameter GateRange with an actual parameter, e.g., the range [1..5]:

```
class 5_gate_reservoir is
      multi_gate_reservoir [1..5]                →
```

Genericity is not limited to ranges: a parameter can be any type, as in a traditional use of genericity whereby one can, for instance, define stacks regardless to the type of elements, etc. Parameters can also be *classes*.

For instance, there could be many kinds of sluice gates, differing for time constants, dimensions, maximum flow. We may wish to describe reservoirs without referring to a particular sluice gate type. This could be done by defining a class *parametricReservoir* with a class parameter *genericGate;* the parameter is employed to declare two modules, *inputGate* and *outputGate*. Then, we could obtain the specification of a particular

Reservoir by providing an actual parameter consisting of a sluice gate of a specific type.

More details on TRIO$^+$ genericity can be found in [MSP].

### 3.3 Inheritance

Inheritance provides the possibility for a class – which in this case is also named *subclass* or *descendant* – to receive attributes from other classes – named its *superclasses* or *ancestors*. The reader can refer to [Mey86] for thorough treatment of inheritance, its many advantages and classifications of the various sorts of inheritance.

The inheritance mechanisms are far from being well settled and universally accepted, probably because their definition is guided by two opposite concepts: *monotonicity* and *freedom*. A monotone approach is characterized by some sort of compatibility between superclasses and subclasses. Conversely, a free approach considers inheritance only a syntactic method to organize classes, without compatibility constraints. This can be achieved in many different ways and degrees, from quasi-monotonicity to total freedom. For example inherited attributes might be redefined only as subtypes of the original ones, or alternatively the user could cancel or redefine them in complete freedom. The reader can find a good discussion of monotonicity in [Win], which defines the terminology we are using.

TRIO$^+$ adopts partially the *monotonicity of the external interface*, like Simula67 or Smalltalk: a visible item must remain visible in the descendants. This avoids the raising of scope errors using inheritance, as we shall see. Other languages, like Eiffel [Mey88], permit to hide inherited attributes from the interface.

Another form of monotonicity tries to impose a behavioral compatibility between superclasses and subclasses: an instance of a class can substitute an instance of its superclasses without any difference in the external behavior. This is very difficult to achieve with reasonable and flexible constraints: it is not practicable to impose not to modify axioms, because they often must change. Eiffel adopts another solution: it tries to force the user to a form of behavioral compatibility, named *monotonicity of the assertions*: Procedures may be redefined, but they must respect some logic assertions, i.e., pre and post conditions and a class invariant.

TRIO$^+$ (like most other object-oriented languages) follows a more liberal approach. We consider behavioral monotonicity too severe for a specification language, which must be flexible. TRIO$^+$ permits to redefine freely the inherited axioms, i.e., to change completely the semantics of the descendants. Also, the feature of the *monotonicity of the data components* is ensured, in that inherited items cannot be canceled (but can be freely redefined). This tries to avoid incorrect uses of inheritance, which can lead to the definition of inconsistent classes.

We now define the essential features of TRIO$^+$ inheritance and of its use for specification purposes.

- A subclass can add and redefine items, modules, and axioms. There is no constraint on the redefinition of axioms and items: an axiom can change to its negation; an item originally defined as a two-place TD predicate can become an unary TI function, etc. The free redefinition of items must be used carefully, because it can make axioms syntactically incorrect. The redefinition of modules can only use a subclass of the original module class. This enforces the monotonicity of the external interface, which avoids the vanishing of visible items of component modules. So all items occurring in inherited axioms are in the current scope too (but their signature can be completely different).
- A subclass can only add connections: it can neither cancel nor modify them. This maintains some correspondence between the internal structures of a class and its superclasses.

An example of axiom redefinition and item addition is the following: define a sluice_gate with an emergency command to open ten times faster than in normal conditions.

**class** sluice_gate_with_emergency_control

**inherit** sluice_gate [**redefine** go, go_up]

**TD Items**

**predicates** go({up, down, fast_up})

**axioms**

*go_up:* (position = down ∧ go(up) ∧
Lasts (¬ go(fast_up), Δ)) →
(Lasts ( position=mvup, Δ) ∧
Futr (position=up, Δ))

*fastup:* position = down ∧ go (fast_up) →
(Lasts (position=mvup,Δ/10) ∧
Futr (position=up,Δ/10))

**end** sluice_gate_with_emergency_control

To change an inherited axiom, item or module, its name must be listed in a redefine clause following the corresponding class (see [Mey88] for the advantages of a similar syntax). In the example, we redefine the item *go*, by adding the new command value *fast_up*, and the axiom *go_up*, and we add the axiom *fastup* to describe the new semantics. For the sake of clarity we have deliberately simplified the axioms: they state that the *fast_up* command has effect only when the sluice is down, and a *go*(*up*) has no effect if a *go*(*fast_up*) will follow within Δ instants. A more realistic behavior would impose to redefine some more inherited formulas. Note that the simple addition of another command imposes the redefinition of some axioms: if such a redefinition was not possible, as in a completely monotone approach, we should define from scratch a new class to describe the new sluice, loosing the advantages of inheritance.

In this example we have followed a natural and incremental style of specification, first defining sluice_gate then specializing it with the class sluice_gate_with_emergency_control. The corresponding inheritance lattice is depicted in Figure 3.4.a. In a different, maybe more orthodox, approach one could define sluice_gate and sluice_gate_with_emergency_control as subclasses of a common "skeleton" class. The corresponding inheritance lattice is depicted in Figure 3.4.b.


**Figure 3.4.a and 3.4.b.** Inheritance lattices of the sluice_gate classes.

Redefinition of modules can be achieved by using a subclass of the class used in the original declaration: for example a reservoir with the new sluice gate for the output is as follows:

**class** reservoir_2nd version
**inherit** reservoir
[**redefine** outputGate, actuator2]
**modules**
outputGate:
sluice_gate_with_emergeny_control;
actuator2:
actuator_with_emergency_control

**end** reservoir_2nd version

The actuator2 must change to control the new sluice gate. For the sake of brevity we do not define its new class.

TRIO$^+$ also allows multiple inheritance. We do not go, however, in the details of its use, referring the interested reader to more technical papers such as [MSP].

## 4. The environment of TRIO$^+$

As we stated before, the practical usefulness of a specification language depends not only on its "conceptual qualities" (rigor, ease and naturalness of use, generality, etc.), but also on the availability of tools that help its use. Thus, we are presently developing a complete environment of TRIO-based tools.

Before describing the main features of such an environment, let us mention that the term "TRIO" also denotes a "family of languages". Besides the basic languages TRIO and TRIO$^+$, the family presently includes two more languages, ASTRAL [GK] and TRIO* [CMR], which can be defined "on top of TRIO$^+$", i.e., as extensions thereof aiming at favoring specialized methodological approaches in system specification. For instance, both ASTRAL and TRIO* favor a specification style based on the notions of *system state* and *transitions* among them.

In general, we believe in an *eclectic approach* to system specification, where the specification language itself should be tailored to a particular application field [GM]. Thus, the TRIO family should be intended as *open*, in the sense that more languages could be added according to specific application features or design methodologies. TRIO$^+$ abstraction mechanisms are particularly well-suited to define language extensions, e.g., as suitable class libraries.

For this reason, we will talk, in the following, of the *environment of the TRIO family*. This is based, essentially, on the following tools:

- Syntax-driven, graphical and interactive editors, which allow the writing of system specifications in different languages of the family. In principle, even "mixed specifications"–consisting of modules coded in different languages–should be possible. We did not yet exploit this feature, however.
- Semantic analyzers. Such tools help specification analysis checking their consistency, completeness, adequacy to user needs. Essentially, these tools are based on interpretation mechanisms. Since, as we saw, TRIO interpretation is computationally intractable or even undecidable in the general case, these tools are based on trade-offs between generality and efficiency (the principle can be roughly stated as follows: "if you want a complex service, you pay for it in terms of efficiency, but if you accept to do some work by yourself–typically, integrating mechanical operations with human intuition–then, you may get more efficiency").
- Synthesis tools. Such tools help the process that leads from specifications to implementation. Thus, they are based on an integration with operational formalisms and programming languages. The use of a language independent formalism, such as Petri nets, allows the construction of a wide portions of these tools in a way that is not affected by the choice of the implementation language, so that the specialization of the tools to cope with different programming languages should be less expensive.
- Implementation verification tools. Such tools help verifying the obtained implementation against the original specifications. Typically, they include test case generators, both of functional and of structural type. More advanced and theoretically oriented tools could help correctness proofs in the same style as classical Hoare's approach to the correctness of Pascal-like programs. See [FMM] for a proof system based on TRIO and timed Petri nets.

Unfortunately, the present state of development of the above environment is still quite in a preliminary stage. We do have, however, significant and useful prototypes of some of the aforementioned tools. Presently they are:

- Editors for the TRIO$^+$ and TRIO* languages. These run on Vax machines equipped with the VMS operating system, but in a very short time UNIX versions will be available.

- Two different interpreters of the basic language TRIO. As mentioned above, one of them is more general but less efficient, whereas the other one exploits some user interaction to achieve more efficiency [FM]. These run on UNIX systems and are presently not yet integrated with the editing tools.

Tools under development are the following:

- Translators from TRO$^+$ and TRIO* to TRIO. These will make executable the specifications written through the above editors.
- Test case editors and generators[1]. The kernel of an automatic test case generator from TRIO specifications is, again, an interpreter thereof, since, roughly speaking, a test case for a system specified through a TRIO formula is a model for the formula itself [MMM].

Other tools are scheduled for longer term development.

## 5. Experiences in the practical application of TRIO$^+$

The TRIO project is a joint research effort of industrial and academic people. From the very beginning, the results of such efforts have been checked against practical applicability in the industrial world, so that many of the languages and tools features are the consequence of the experience derived from early applications. Some observations derived from this experience are perhaps worth mentioning, since much debate is still ongoing on the practical applicability of formal methods.

Several cases studies have been performed using TRIO$^+$ to specify real life systems to be developed by the industrial partners of the project. These case studies included the specification of a pondage power system controller, of an electric energy distribution system, and of several hardware components. Detailed documentation on these case studies can be found in [MSP, CMS].

In all cases the results of the specification work was universally agreed to be of far higher quality (mainly in terms of precision and clarity) than what was obtained by classical informal (natural language documentation) or semiformal (Data-flow Diagrams) methods.

In some case, the use of semantic analysis tools such as prototypers allowed the discovering a few subtle specification errors that remained uncovered by human inspection.

These successes, however, were obtained either directly by the developers of the language or by other people who were already used to cooperate with them.

Not as successful, instead, has been the experience of teaching new people the autonomous use of the languages. This effort is still ongoing and final results are still to be obtained. In our opinion, this is due to two different reasons.

On the cultural side, managing mathematical formulas of any type requires much training, mainly for people whose university curriculum is not quite recent. On this respect, we realized that, in general, the use of operational formalisms such the old fashioned finite state machines, but even the more powerful Petri nets is easier to learn for industrial people than descriptive formalisms such as those based on algebra or mathematical logic.

On the organizational side, we much suffered from the fact that the managing gave lower priority to training activity than to the production activity. Thus, many scheduled meetings have been suddenly canceled and many delays occurred, in spite of the fact that the involved people were quite willing and anxious to learn the new language, even at the price of working on it during the nights and the weekends–i.e., gratis!, perhaps because they were really convinced of the potential benefits of the new tools.

In summary, we may synthesize the lessons learned from this early experience in the application of TRIO in the industrial world with the following "tale", which should be well applicable to formal methods in general:

[1] By "test case editor" we mean a tool that helps the user to write good testing documentation; by "test case generator" we mean a tool that (semi)automatically builds test cases from specifications (functional testing) or from implementation code (structural testing).

Initially, the academic world believed that its job was producing new and exciting ideas and that their exploitation in practice was just someone else's–trivial–job.

This turned out to be a big mistake because many difficulties had still to be overtaken in the long path from theory to practice. The result was the failure of the proposed methods and the skepticism of the industrial world.

Then, it was realized that it was necessary to demonstrate practical applicability of new solutions to real-life, non toy, cases. This has been done, in general, with good success (we may find now in the literature reports on several successful applications of formal methods to real-life problems.)

Again, academicians thought that their job was ended with the publication of such reports; but, again, very few industrial people, left alone with the new formalisms, were able to apply them by themselves and eventually discarded it.

The conclusion of the story is twofold: on the one side, the academic world should be ready to devote even more effort to technology transfer towards the industry than to scientific originality; on the other side the industrial world should be genuinely willing to put much effort in an endeavor that cannot produce "miracles", i.e., great increases in productivity at lowest costs in shortest times.

## Acknowledgments

As mentioned above, the TRIO project is a joint effort of university (Politecnico di Milano) and industry (ENEL[1]-CRA and CISE). The research is mainly funded by ENEL-CRA with some support also from CNR (Italian National Research Foundation). The project started in 1988. Since the starting of the project, several people have contributed to this research, whose principal investigator is the author of this paper. From Politecnico, let us mention Miguel Felder, Carlo Ghezzi, Sandro Morasca, Angelo Morzenti, Pierluigi SanPietro. From CISE, Emanuele Ciapessoni, Edoardo Corsetti, Manlio Migliorati, Angelo Montanari, Elena Ratto, Marco Roncato, Luigi Zoccolante. From ENEL-CRA, Ernani Crivelli, Valeria Filippi, Giovanni Maestri, Roberto Meda, Piergiorgio Mirandola. Recently, Michel Piguet from EDF (the french agency of energy) joined the group.

TRIO$^+$ has been mainly developed by Pierluigi SanPietro e Angelo Morzenti. This paper owes much to their original paper [MSP] (with authors' consensus).

In a fairly independent way, the language ASTRAL has been developed by Carlo Ghezzi and Richard Kemmerer while Ghezzi was on sabbatical leave at UCSB [GK].

It is group's policy that all project results are public domain. Thus, report, tools, and manuals can be obtained either from ENEL-CRA or from Politecnico.

[1]ENEL is the italian agency of energy.

## References

[BjP] Bjorner D., Prehn S.,
Software engineering aspects of VDM, the Vienna Development Method
In *Theory and Practice of Software Technology*, Ferrari D. et al., editors, pp 85-134, North-Holland, Amsterdam, 1983.

[CHB] Coleman D., Hayes F., Bear S.
Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design
*IEEE Transactions on Software Engineering* , Vol. 18, N. 1, Jan. 1992, pp. 9-18

[CHJ] Cohen B., Harwood W.T., and Jackson M.J.
*The Specification of Complex Systems,*
Addison Wesley, Reading (1986).

[CMR] Corsetti E., Montanari A., Ratto E.
A Methodology for an Incremental, Logical Specification of Real-time Systems
Proc. 2nd Euromicro Workshop on Real-Time Systems, Horholm, Denmark, May 1990

[CMS] Coen A., Morzenti A., Sciuto D.
Hardware Specification with the Temporal Logic Language TRIO
1990 ACM Int. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Vancouver, Canada, October 1990

[DeRK] DeRemer F., Kron H.
Programming-in-theLarge versus Programming-in-the-Small.
*IEEE Transactions on Software Engineering* Vol. 2, N. 6, June 1976, pp. 80-86.

[FM] Felder M. and Morzenti A.
Validating Real-Time Systems by executing logic specifications in TRIO.
In Proc.*14 IEEE/ACM International Conference on Software Engineering,* 1992.

[FMM] Felder M., Mandrioli D., Morzenti A.
Proving properties of real-time systems through logical specifications and Petri net models
Internal Report, Dipartimento di Elettronica, Politecnico di Milano, 1991. Submitted for publication.

[GJM] Ghezzi C.,Jazayeri M., Mandrioli D.
*Fundamentals of Software Engineering*,
Prentice-Hall, Englewood Cliffs, 1991.

[GK] Ghezzi C., Kemmerer R. A.
ASTRAL: an Assertion Language for Specifying Real-Time Systems
Proc. 3rd ESEC, Milano, Italy, October 1991.

[GM] Ghezzi C., Mandrioli D.
On Eclecticism in Specifications: A Case Study centered around Petri Nets.
In *Advances in Object-Oriented Software Engineering*, Mandrioli D., Meyer B. (editors), Prentice-Hall, 1992.

[GMM] Ghezzi C., Mandrioli D., and Morzenti A.
TRIO, a logic language for executable specifications of real-time systems.
*Journal of Systems and Software* Vol. 12, N. 2, May 1990, pp. 107-123.

[Har et al] Harel D., Lachover H., Naamad A., Pnueli A., Politi M., Sherman R., Shtull-Trauring A., Trakhtenbrot M.
"STATEMATE: A working environment for the development of complex reactive systems", *IEEE Transactions on Software Engineering*, Vol. 16, N. 4, April 1990, pp. 403-414.

[Kem] Kemmerer R. A.
"Testing formal specifications to detect design errors"
*IEEE Transactions on Software Engineering*, Vol. 11, N. 1, January 1985, pp. 32-43.

[MBM] Mili A., Boudriga N., Mili F.
*Towards strucured specifying: theory, practice, applications*
Ellis Horwood Ltd, Chichester, England, 1989.

[Mey86] Meyer B.
Genericity versus Inheritance.
In Proc.*OOPSLA*, Portland, Oregon, 1986.

[Mey88] Meyer B.
*Object Oriented Software Construction*
Prentice Hall, 1988.

[MMG] Morzenti A., Mandrioli D., and Ghezzi C.
A Model Parametric Real-Time Logic.
*ACM Transactions on Programming Languages and Systems*, July 1992.

[MMM] Mandrioli D., Morasca S., Morzenti A.
Functional Test Case Generation for Real-Time Systems
Proc. Third Conf. on Dependable Computing for Critical Applications Palermo, Italy, September 1992.

[MSP] Morzenti A., San Pietro P.
Object-Oriented Logic Specifications of Time Critical Systems
Internal Report, Dipartimento di Elettronica, Politecnico di Milano, 1991.

[Ost] Ostroff J.
*Temporal Logic For Real-Time Systems*,
Research Studies Press LTD, Taunton, Somerset, England , Advanced Software Development Series, 1989.

[Pet] Peterson J.L.
*Petri Net Theory and the Modelling of Systems*
Prentice-Hall, Englewood Cliff NJ, 1981.

[ShM] Shlaer S., Mellor S
An Object-Oriented Appraoch to Domain Analysis.
In *Advances in Object-Oriented Software Engineering*, Mandrioli D., Meyer B. (editors), Prentice-Hall, 1992.

[Spi] Spivey J.
*The Z Notation–A Reference Manual*
Prentice-Hall, Englewood Cliffs, N.J., 1989.

[Win] Winkler J.F.H.
Square IS-A Rectangle? or IS-A Inheritance in OOP and OOA/D. Manuscript.