An Object-Oriented Logic Language for Modular System Specification °

Angelo Morzenti and Pierluigi San Pietro
Politecnico di Milano, Dipartimento di Elettronica, Piazza Leonardo da Vinci 32, Milano, Italy.

Abstract

We define TRIO$^+$, an Object Oriented logic language for modular system specification. TRIO$^+$ is based on TRIO, a first order modal language that is well suited to the specification of embedded and real-time systems, and provides an effective support to a variety of validation activities, like specification testing, simulation, and property proof. Unfortunately, TRIO lacks the possibility to construct specifications of complex systems in a systematic and modular way. TRIO$^+$ combines the use of constructs for hierarchical system decomposition, and object oriented concepts like inheritance and genericity, with an expressive and intuitive graphic notation, yielding a specification language that is formal and rigorous, and still flexible, readable, general, and easily adaptable to the user's needs.

1 Introduction

The importance of the requirement specification phase for the development process of any system, and of software and hardware systems in particular, has been greatly emphasized in recent years. In fact, incorrect, incomplete, or poorly structured specifications can cause significant organizational and economical problems in all successive phases of system development. This justifies the great attention devoted in the research community to the study of specification languages and methods.

Formal specification methods proved to be well suited to the production of rigorous and unambiguous specifications. Formality in specifications also allows one to perform systematically or even automatically validation activities like testing and simulation, and to prove that the modelled systems possess desired properties. The use of formal methods is particularly valuable in the most "difficult" areas, such as embedded and real-time systems: these systems are required to perform critical or dangerous tasks, interacting with an environment which evolves independently at a speed that cannot be controlled. Typical examples in this category are weapon systems, patient control systems, plant control systems, flight control systems, etc: a failure of the system to react to certain input data signals within some specified time bounds can cause severe damages or even fatal disasters.

As [Wir 77] points out, "time" plays a fundamental role in real–time systems; in particular, correctness of such systems does depend on time. This constitutes a sharp departure from sequential and concurrent systems. In the case of sequential systems, time simply affects performance, not correctness. In the case of concurrency, systems can always be designed in a way that their behavior (and thus, correctness) does not depend on the speed of activities. Unfortunately, this is not true in the case of real–time systems, since ac-

° This work was partially supported by CNR-Progetto Finalizzato Sistemi Informativi e Calcolo Parallelo and by ENEL-CRA.

tivities occurring in the environment are not entirely under control. They evolve according to their own logic: they cannot be delayed and resumed "ad libitum" to meet the desired correctness criteria.

In the past years we developed TRIO [GMM 90], a logic language for the formal specification of real-time systems. TRIO is based on classical temporal logic, but it deals with time in a quantitative way by providing a metric to indicate distance in time between events and length of time intervals. Another extremely important aspect is that TRIO's formal semantics can accommodate a variety of time structures: from dense to discrete and finite. In particular, finite time structures are those that will be used to execute TRIO specifications. TRIO is provided with a formal semantics which permits validation activities such as testing specifications against a history of the system evolution, simulation of the system behavior in response to a sequence of stimuli coming from the environment, and formal proof of system properties. In [MRR 89] it was shown how the TRIO language can become the core of a specification environment where suitable tools can provide an automatic support to the specifications activity.

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. However, the use of TRIO for the specification of large and complex systems 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.

However in the description of large and complex systems [CHJ 86], 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 89]. These issues are similar to those arising in the production of large programs, an activity that is usually called programming-in-the-large [D&K 76]. Hence we may refer to the process of producing specifications of complex systems as specifying-in-the-large.

To support specification in the large, we enriched TRIO with 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 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 other parts that are considered less important or 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 of the 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, expecially when the graphic view is consistently supported by means of suitable tools, such as structure-directed editors, consistency checkers, and report generators.

In our opinion this is the reason of the popularity of the so-called CASE tools, many of which are based on Data Flow Diagrams or their extension [DeM 78, Y&C 79, War 86]. These tools comprise informal or semi-formal languages as their principle descriptional notation, and exhibit problems such as ambiguity, lack of rigor, and difficulty in executing specs, but nevertheless they can be very helpful in organizing the specifier's job. On the other hand TRIO$^+$ aims at providing a formal and rigorous notation for system specification, which includes effective features and constructs to support modularization, reuse, incrementality and flexibility in the specification activity.

The paper is organized as follows. Section 2 summarizes TRIO's main features, provides a simple model-theoretic semantics, and describes how the language can be used to perform validation activities on the specifications. Section 3 introduces TRIO$^+$, the object oriented extension of TRIO; in particular, it illustrates the constructs for inheritance and genericity. The concepts are mostly presented through examples, and whenever possible a graphic representation of the specifications is constructed in parallel with the textual one. The semantics of TRIO$^+$ is not provided in full detail: only an informal and sketchy description is given of how TRIO$^+$ specifications can be translated into suitable TRIO formulas. Section 4 draws conclusions and indicates some directions of future research.

2 Definition of the TRIO language

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 90].

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 '$\wedge$', the derived ones '$\rightarrow$', '$\vee$', '$\leftrightarrow$', ..., 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 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, or a subset thereof. 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

$$\text{Futr}(s, t) \qquad \text{and} \qquad \text{Past}(s, t)$$

are also TRIO terms. The intended meaning is that, if *v* is the numerical value of term *t* , then the value of *Futr (s, t)* (resp. *Past (s, t)*) is the value of term *s* at an instant lying *v* 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

$$\text{Futr}(A, t) \qquad \text{and} \qquad \text{Past}(A, t)$$

are TRIO formulas too that are satisfied at the current time instant if and only if property *A* holds at the instant lying *v* time units ahead (resp. behind) the current one. Based on the primitive temporal operators *Futr* and *Past*, numerous derived operators can be defined for formulas. We mention, among the many possible ones, the following:

$$\begin{array}{lll}
\text{AlwF}(A) & \overset{\text{def}}{=} & \forall t\,(t > 0 \rightarrow \text{Futr}(A, t)) \\
\text{AlwP}(A) & \overset{\text{def}}{=} & \forall t\,(t > 0 \rightarrow \text{Past}(A, t)) \\
\text{Lasts}(A, t) & \overset{\text{def}}{=} & \forall t'\,(0 < t' < t \rightarrow \text{Futr}(A, t')) \\
\text{Always}(A) & \overset{\text{def}}{=} & \text{AlwP}(A) \wedge A \wedge \text{AlwF}(A) \\
\text{NextTime}(A, t) & \overset{\text{def}}{=} & \text{Futr}(A, t) \wedge \text{Lasts}(\neg A, t) \\
\text{Becomes}(A) & \overset{\text{def}}{=} & A \wedge \exists \delta\,(\delta > 0 \wedge \text{Past}(\neg A, \delta) \wedge \text{Lasted}(\neg A, \delta))
\end{array}$$

AlwF(A) means that *A* will hold in all future time instants; *Lasts(A, t)* means that *A* will hold for the next *t* time units; *NextTime(A,t)* means that *A* will take place for the first time in the future at a time instant lying *t* time units from now; *AlwP* has, for the past, the same meaning than the corresponding operator for the future. *Always(A)* means that *A* holds in every time instant of the temporal domain.

As an example, we consider a pondage power station, where the quantity of water held in the basin 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 Δ time units to go from the *down* to the *up* position, after receiving a go (up) command.

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

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.

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

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.

In a way similar to what is done in classical first order logic, one can define the concepts of satisfiability and validity of a TRIO formula, with respect to suitable interpretations. For the sake of simplicity, in the following, we provide a straightforward model theoretic semantics, that can assign meaning to TRIO formulas with reference to unbounded temporal domains, like the set of integer, rational or real numbers. In [MMG 90] a truly model-parametric semantics for the language is defined, which may refer to any finite or infinite temporal domain. An interpretation for a TRIO formula is composed of two parts: a time dependent part L and a time independent one, G. The time independent part, $G = (\xi, \Pi, \Phi)$ consists of a variable evaluation function, ξ, that assigns values to time independent variable names, of a time independent predicate evaluation function, Π, that assigns a relation to every time independent predicate name, and of a function evaluation Φ that assigns a function to every function name. The time dependent part $L = \{(\xi_i, \Pi_i, \Phi_i) \mid i \in T\}$ contains, for each instant in the temporal domain T, one evaluation function ξ_i for time dependent variables, one evaluation function Π_i that assigns a relation to every time dependent predicate name, and one evaluation function Φ_i that assigns a function to every time dependent function name. Based on such interpretation, it is possible to define an evaluation function assigning a value to all TRIO terms and formulas in a generic time instant $i \in T$. For a complete definition of the model-theoretic semantics of TRIO, the interested reader can refer to [M&S 90b] which contains an extended version of the present paper.

A TRIO formula is said to be temporally *satisfiable* in an interpretation if it evaluates to true in at least one instant of the temporal domain. In such a case we say that the interpretation constitutes a *model* for the formula. A formula is said to be temporally *valid* if it is true in every instant of the temporal domain. Finally, a TRIO formula is said to be *time invariant* if it is either valid or it cannot be satisfied in any interpretation.

A TRIO formula is *classically closed* if all of its (time independent) variables are quantified; it is *temporally closed* if it does not contain time dependent variables or predicates, or if it has either *Sometimes* or *Always* as the outermost operator, or finally if it results from the propositional composition or classical closure of temporally closed formulas. It can be proved (see [Mor 89]) that any temporally closed formula is time invariant; this can be understood intuitively by considering that the operators *Sometimes* and *Always* provide a way to quantify existentially and universally the current time which is implicit in TRIO formulas.

For these reasons we define a *specification* of a real time system as a TRIO formula that is closed, both classically and temporally.

Since TRIO is an extension of the first order predicate calculus, it is evident that the problem of the satisfiability of a TRIO formula is undecidable, in its full generality. The problem is however decidable if we consider only interpretations with finite domains. In this hypothesis, we defined an algorithm that proves the satisfiability of a TRIO formula on a finite interpretation in a constructive way, that is, by constructing an interpretation where the formula is verified. This algorithm was inspired by the tableaux proof methods, that were first defined in [Smu 68] and have been widely used in different branches of temporal logic, like in [R&U 71, Wol 83, BPM 83]; the interested reader may refer to [GMM 90] for its detailed definition. The algorithm for the proof of the satisfiability of a formula can be used to prove its validity, by showing that its negation is unsatisfiable; also we can show that a specification ensures some given property of the described system, by just proving as valid the implication $\Sigma \rightarrow \Pi$, where Σ is the specification, and Π is the desired property. Thus, the specification can then be said to be executable, since we can perform proofs of properties in a mechanical way.

We point out that the tableaux-based algorithms, that permit the execution of TRIO specifications by constructing interpretations for closed formulas, assume as given the time independent part of the interpretation which assigns values to time independent predicate and functions. This because such logic entities represent static and general facts or invariant properties of the modelled system that are well known to the specifier. By contrast, when executing the specification, one is more interested in constructing the time dependent part of the interpretation, since it represents the events taking place in one possible dynamic evolution of the system.

Furthermore, executability of TRIO formulas is also provided at lower levels of generality, by giving the possibility to verify that one given temporal evolution of the system is compatible with the specification, and to simulate the specified system, starting from an initial, possibly incompletely specified, configuration. See [GMM 90, F&M 91] for more details about simulation and verification with TRIO.

3 Definition of the TRIO+ language

As we showed in section 2, the TRIO specification of a system is built by writing logical axioms. The TRIO$^+$ specification of the same system is expressed defining suitable *classes*. A class is again a set of axioms describing the system, but this set is built up in a modular, independent way, following information hiding principles and object oriented techniques. Classes may be *simple* or *structured*, and can be part of an *inheritance* lattice.

3.1 Simple classes

A simple class is very similar to an ordinary TRIO specification: it is a group of axioms, in which occurring predicates, variables, and functions must be explicitly declared, in order to have typed formulas. An example of simple class is the specification of the sluice-gate already treated in the example of section 2.

class sluice_gate
  Visible go, position
  Items   go: $TD \times \{up, down\} \rightarrow boolean$
        position: $TD \rightarrow \{up, down, mvup, mvdown\}$

Δ: $\rightarrow$ integer

Vars t: integer

Axioms

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

go_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

The class header 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 keyword **Items** is followed by the declarations of the local functions, predicates and variables which can be used in the axioms. The declarations are based on predefined scalar types, such as integer, real, boolean, finite sets, subranges. In the example Δ is an integer constant (constants are declared as zero-ary functions), *go* is a unary time dependent predicate on the set {up, down} (predicates are declared as boolean functions), *position* is a time dependent variable whose values may range on {up, down, mvup, mvdown} (time dependent variables are declared as a function of Temporal Domain, TD). Items are time dependent if they have TD as the type of their first argument; however, no corresponding argument appears in the use of the identifier in the formula, since time is implicit in TRIO. The **Vars** clause is followed by the declaration of the time independent variables which occur in the axioms. The **Axioms** are TRIO formulas, prefixed 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:

Always(position = down $\wedge$ go (up) $\rightarrow$ Lasts (position = mvup, Δ) $\wedge$ Futr (position = up, Δ))

Every axiom can be preceded by a name, which can be used as a reference for axiom redefinition in inheritance: see section 3.5. The name must be different from the names of the items of the class. The items 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 relaxed in section 3.2.

An **instance** of the 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. As in the execution of TRIO specifications, we are mainly interested in the generation of the dynamic (i.e. time dependent) part of the interpretation, and assume that the static (i.e. time independent) part is given. A class instance of sluice_gate is then the following table (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 1. An instance of the class sluice_gate

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); Δ instants after the command, the gate is up (position=up).

TRIO$^+$ is a pure logic language: no surprise there are *no* primitives like Create or New to explicitly control instance creation.

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 is continued outside the box. Class *sluice_gate* is represented in Figure 1.

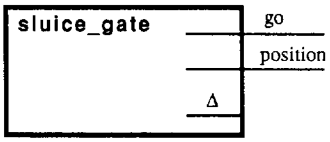

Figure 1. Graphic representation of the class sluice_gate

3.2 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 which are instances of those classes. For example, a tank 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, to represent 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, expecially suited to describe systems in which *parts*, i.e. *modules*, can be easily recognized. The tank of the above example 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 tank -- first partial version --
        Visible inputGate.go, outputGate.go
        Modules inputGate, outputGate: sluice_gate
end tank
```

Recursive definitions of classes are not allowed: so a class (and its subclasses: see section 3.5) can not be used to declare its own modules. Structured classes have a meaningful graphic representation: the modules of the class are just boxes, with a name and a line for every visible item. The picture for the tank example is in Figure 2.

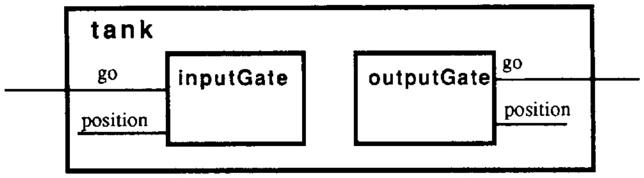

Figure 2. The graphic representation of class tank (draft version)

Modules cannot be used directly in axioms, because they are not logical symbols such as predicate or function names: they represent a set of items and modules definitions, with related axioms. For the same reason the visible interface can not 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 *tank*, 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*.

3.3 Specifying complex systems

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. A more realistic tank may have two actuators, one to control each sluice gate, and a transducer which measures the level of the tank. The external plant is able to send four commands to control the tank, to open or close each sluice gate. This can be described by defining the class *tank*, depicted in Figure 3.

```
class tank
    Visible transducer.level, openInput, closeInput, openOutput, closeOutput
    Modules       inputGate, outputGate: sluice_gate
                  transducer: cl_transducer
                  actuator1, actuator2: actuator
    Items openInput, closeInput, openOutput, closeOutput: TD → Boolean
    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) }
end tank
```

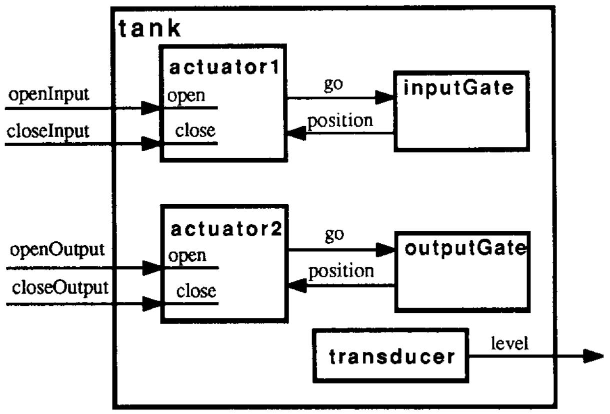

Figure 3. The graphic representation of the class *tank* (final version).

Every instance of this class contains two instances of sluice_gates, one instance of *cl_transducer* and two 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 which are 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 items connected: it is only an expressive, although informal, notation.

In the example, we state that the commands openInput, closeInput, openOutput, closeOutput and the instantaneous value of tank level are all what the external world is allowed to know of a tank. The connections say 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 can be a useful method for the specification of a complex system: one describes system components separately and then identifies information flows between them. Using connections this can be made in a systematic way, with an expressive and convenient graphic meaning. A structured class can be thought as a complex system, composed of interacting subsystems; interactions are shown by the connections between modules.

Another TRIO$^+$ facility is a tool to describe those real world 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, and so on. These situations are easily described in TRIO$^+$, because it is possible to define *arrays* of modules. For example, a class for a four-bit shift register may be declared as in the following, using a separately defined class DT_flip_flop:

```
class 4_places_shift_register
    ....
    Modules DT: 1..4 → DT_flip_flop     -- an array of four modules, which are accessed as DT(.) --
    ....
end 4_places_shift_register
```

3.4 Genericity

TRIO$^+$ is provided with a simple genericity mechanism (see [Mey 86] for a good introduction to genericity in object oriented languages). Generic classes have one or more parameters, which must be instantiated before use: generic classes are not directly executable. Parameters can be types for the items or classes for the modules, but also constants which can be used to declare subranges.

For example, a generic n-elements shift-register can be so declared:

```
class N_places_shift_register [NumEl]    -- NumEl is a constant parameter --
    ....
    Modules DT: 1..NumEl → DT_flip_flop
    ....
end N_places_shift_register
```

To declare an eight-place shift-register one can write:

```
class 8_places_shift_register is N_places_shift_register [8]
```

The same syntax can be used to declare a class is generic with respect to classes or types.

3.5 Inheritance

Inheritance provides the possibility for a class–which in this case is also called a *subclass*–to receive attributes from other classes–called its *superclasses*. The reader is referred to [Weg 88] for thorough treatment of inheritance, its many advantages and a classification of the various sorts of inheritance.

The inheritance mechanisms are far to be well settled and universally accepted, but their definition is guided by two opposite concepts: monotonicity and freedom. The monotone approach is characterized by a strict semantic compatibility between superclasses and subclasses: every instance of a class must then be an instance of its superclasses, since it must satisfy all their axioms. This is very difficult to achieve in practice with reasonable and flexible constraints. The other approach, which is much more common, is to consider inheritance only a syntactic method to organize classes. This can be achieved in many different ways and degrees, from quasi-monotony to total freedom. For example inherited attributes might be redefined only as subtypes, or alternatively the user could be allowed to cancel or redefine them in a completely free way.

In TRIO$^+$ inheritance follows the liberal approach, because monotonicity is considered too severe for a specification language, which must provide a good degree of flexibility. Our definition tries to avoid incorrect uses of inheritance, which can lead to the definition of inconsistent classes, imposing some constraints. TRIO$^+$ inheritance allows the specifier to add and redefine items, modules, and axioms. The axioms can be redefined in a totally free way, while the redefinition of items and modules has some restrictions. Connections are inherited without changes. Redefinition of items is free, but users cannot change their arity, e.g. a two-place predicate must remain two-place, or else all formulas where it occurs would become syntactically incorrect. Instead, the declaration of the domains of items can change freely.

An example of axioms 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]
  Items go: TD $\times$ {up, down, fast_up} $\rightarrow$ boolean
   Axioms
   go_up: position = down $\wedge$ go(up) $\wedge$ Lasts ($\neg$ go(fast_up), Δ) $\rightarrow$
      Lasts (position=mvup, Δ) $\wedge$ Futr (position=up, Δ)
   fastup: position = down $\wedge$ go (fast_up) $\rightarrow$ Lasts (position=mvup, Δ/10) $\wedge$ Futr (position=up, Δ/10)
end sluice_gate_with_emergency_control

In order to change an inherited axiom, item or module, its name must be listed in a redefine clause following the corresponding class (see [Mey 88] for the advantages of a similar syntax). In the example, we redefine the item *go*, 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 that 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.

Notice that the simple addition of one more command imposes the redefinition of some axioms: if such a redefinition was not possible, as in a monotone approach, we should define from scratch a completely new class to describe the new sluice, losing the advantages of inheritance. Redefinition of modules must be achieved by specifying a subclass of the class used in the original declaration: for example a tank with the new sluice gate for the output is obtained as follows:

class tank_2ndversion
  Inherit tank [**redefine** outputGate, actuator2]
  Modules outputGate: sluice_gate_with_emergeny_control
    actuator2: actuator_with_emergency_control
end tank_2ndversion

The actuator2 must change to control the new sluice. For the sake of brevity we do not define its new class. A good solution to minimize redefinitions is to use multiple inheritance.

The constraint of using only subclasses to redefine modules is imposed in order to avoid that visible items of the component modules vanish, because the new module class used for redefinition does not possess them: this could make axioms incorrect, because they would refer to items that do not exist in the scope of the heir class. For the same reason we forbid to make invisible the inherited visible items: to this end, it would be equivalent to a cancellation.

The definition of **multiple inheritance** brings no additional difficulties, since it only requires to solve name clashes of inherited attributes that are homonymous. Using the technique adopted in the programming language Eiffel [Mey 88], name clashes are avoided by the use of a *rename* primitive, to differentiate inherited attributes. Note that there is no name clash if the two attributes are inherited from a common superclass, and they have not been redefined in an intermediate ancestor. An example of this situation and of the use of inheritance to classify and incrementally specify systems is the following description of S-R and J-K flip-flop's, starting from more elementary devices, according to the inheritance lattice shown in Figure 4.

All flip-flop's have an output and a delay of propagation between the commands and their effect. Thus a general definition of flip-flop's would be:

```
class flip_flop
    Visible Q
    Items     Q: TD → Boolean     -- the state of the flip-flop, which can be True or False --
              τ: → Real           -- maximum time of propagation of flip_flop --
end flip_flop
```

This class is so simple and general it has no axioms. A first specializations is adding a Set command:

```
class Set_ff
    Inherit flip_flop
    Items S: TD → Boolean
    Axioms Set : S → Futr(Q,τ)        -- a set command makes Q true after τ instants --
end Set_ff
```

Another possible specializations is adding a Reset (or *Clear*) command:

```
class Reset_ff
    Inherit flip_flop
    Items R: TD → Boolean -- R stands for Reset --
    Axioms Reset:  R → Futr(¬Q,τ) -- a Reset command makes Q false after τ instants --
end Reset_ff
```

Now we want to describe a S-R flip-flop: it has both Set and Reset, and the output persists in its value when there is no command; it is not contemplated the possibility of Set and Reset true at the same time.

```
class SR_ff
    Inherit Set_ff, Reset_ff
      -- there is not a name conflict for τ, because it is inherited from a unique superclass: flip_flop --
    Axioms Persistency: ¬S ∧ ¬ R →( (Q → Lasts(Q,τ)) ∧ (¬Q → Lasts(¬Q,τ)))
end SR_ff
```

The *Persistency* axiom assures that in absence of any command the value of the output remains unchanged. A JK flip-flop (see Figure 5) is like SR but allows simultaneous Set and Reset (called J and K), which has the effect to change the state, whatever it is. A class describing JK's is easily obtained by specialization of SR_ff. Now the two axioms of set and reset must change:

class JK_ff

Inherit SR_ff **[rename** S **as** J, R **as** K**] [redefine** Set, Reset**]**

-- the renaming is not for a name clash, but only because J and K are more frequent names for S and R --

Axioms *Set*: $J \wedge \neg K \rightarrow \mathrm{Futr}(Q,\tau)$

Reset: $K \wedge \neg J \rightarrow \mathrm{Futr}(\neg Q,\tau)$

Commutation: $J \wedge K \rightarrow (Q \leftrightarrow \mathrm{Futr}(\neg Q,\tau))$

end JK_ff

Note that J and K are visible even if this is not stated explicitly in the class JK_ff: they were visible in the superclasses S_ff and R_ff.

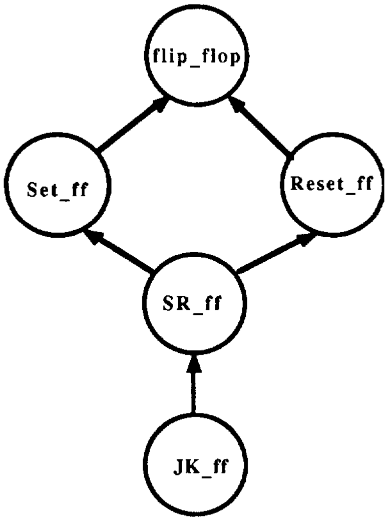

Figure 4. The inheritance lattice for the flip-flop's

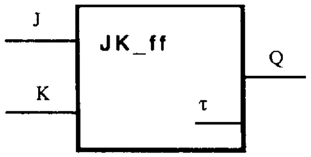

Figure 5. The picture of class JK-ff

The example shows once more that even for simple objects the specification of a subclass may need retracting superclasses' axioms. When defining the set and reset commands one thinks there is no exception: a *set* command will always make the output true, and similarly for a *reset*. This is still true for SR flip-flop's. But when specifying JK flip-flop's one discovers that sometimes *set* is not sufficient to guarantee the output to be true: if at the same time a *reset* happens, then the output can become true or false, depending upon its past value. So one needs to change the set and reset axioms.

3.6 Semantics of TRIO$^+$

The semantics of TRIO$^+$ is provided via translation of a set of class declarations into a simple closed TRIO formula. Such formula is to be intended as a semantically equivalent (although poorly structured!) TRIO specification of the system originally described in TRIO$^+$. A formal and thorough treatment of the semantics of TRIO$^+$ is beyond the scope of the present paper; for this reason, and for the sake of brevity, in the following we will just provide an informal and descriptive account of the translation process, giving an intuitive explanation of its main steps and leaving to the reader's intuition the burden to fill in the missing details.

Since the axioms in TRIO$^+$ specifications are ultimately expressed in terms of the *items*, which are the elementary, indivisible parts, the translation can be performed quite easily, by rephrasing the axioms in terms of the smallest components of the system, which can be uniquely identified by means of a suitable naming convention. In case the system contains arrays of homogeneous modules, the naming convention will take this fact into account by adding a suitable number of arguments to the corresponding TRIO entities: constants will be translated into functions, and functions and predicates will acquire additional arguments. The translation of a TRIO$^+$ specifications into a TRIO formula is described by the following sequence of five steps.

1. Substitute textually the actual parameters (constant values or class names) in place of the formal parameters of generic classes. Include (again by means of textual substitutions) the parts (i.e. both items and modules) inherited and not redefined in all descendant classes. The result of these substitutions is a set of class declarations deprived of any form of genericity or inheritance.
2. Translate all *connections* clauses into the corresponding equivalence (for connected boolean parts) or equality (for other kinds of connected parts) axioms. We recall that connection clauses are allowed only among items (local, inherited or imported) visible in the current class declaration.
3. Determine the alphabet of the TRIO formula to be obtained as translation of the class specifying the system. Starting from the set of class declarations, consider the class that represents the specified system, that is, the highest class in the compositional hierarchy, and construct the *tree of parts* of its instances. The root of the tree corresponds to the class instance itself, the leaves are the *item* components of all involved classes, and each intermediate node corresponds to the module components of the structured classes. A node corresponding to a component of a given class has one leaf child for every item component, and one non-leaf child for every module component. By associating to the root node the empty string "", and to every other node the name of the part it represents[1], each leaf of the tree can be uniquely identified by the concatenation of the strings associated to the path from the root to the leaf itself.
   The type of each element of the alphabet is the type of the corresponding item, if none of the components of which the item is a (sub)part are functions with a TRIO$^+$ class as range, i.e. if the item is not included in any array of homogeneous parts. Otherwise, in presence of a declaration of an array of parts, the element of the alphabet acquires as arguments those of the function used to define the array. Thus, if the item was defined in TRIO$^+$ as an n-ary function or predicate, then the addition of new arguments augments its arity; otherwise, if the item is a constant, then it becomes a function whose arguments are the indices in the array of components.
   Any element of the resulting TRIO alphabet will be a time dependent entity if and only if this was the case for the TRIO$^+$ item it represents.

[1] Recall that all parts have distinct names, and that parts of a class are *not* classes.

4. Translate every axiom in the class definitions into a TRIO formula, proceeding top-down from the system specification to its components, subcomponents and so on. In the axioms, substitute the names of the items with the names of the corresponding leaves in the tree of parts, with possible additional arguments as determined at step 3. The added variables corresponding to indices of vectors of components must be universally quantified, since the axioms of the corresponding class must be satisfied by every component. The axioms will be nested according to the structure of the tree of parts, and the scope of each added variable will be the set of axioms of the descending parts.
5. Conjunct all axioms and close temporally with an *Always* operator the resulting formula, thus obtaining the desired translation of TRIO$^+$ into TRIO.

Any model of the TRIO formula represents an object, or instance, of the class defined intensionally by the TRIO$^+$ declarations. As stated in section 3.2, the components of a TRIO$^+$ instance are instances of the respective classes. In logical terms, their models are constituted by the parts, in the overall TRIO structure, which assign a value to the corresponding elements of the TRIO alphabet, generated as in step 3. Let us consider on a toy example the result of the translation process of TRIO$^+$ specifications into TRIO formulas. We define a system like the tank of sec. 3.2, but with an array of level sensors. Every sensor is placed at a suitable depth in the tank and has a two-value output, which we represent with a time dependent propositional variable *above*, to indicate whether the water is or is not above the sensor. We assume that the value of *above* can be altered only after at least three seconds from the last change.

```
class sensor
      Visible above
      Items        above: TD → Boolean
      Axioms
              min-period: Becomes(above) → Lasts(above,3) ∧ Lasted(¬above,3)
end sensor
```

The sluice gates of the tank are of the class *sluice_gate* defined in Section 3.1, but for the sake of simplicity in the following we will only one of its axioms, *go_down*:

go_down: position=up ∧ go(down) → Lasts (position=mvdown, Δ) ∧ Futr (position=down , Δ)

The tank includes three sensors, which are placed respectively at low, middle, high levels, indicating thus four possible situations for the level of the water: below all sensors, between low and middle, between middle and high, above all sensors. All other situations are incorrect and must be signalled. The array of sensors can be represented as a function from the enumerated type *(low, middle, high)* to the class *sensor*.

```
class tank
      Visible        SensorFault, InputGo, OutputGo
      Items          SensorFault: TD → Boolean
                     InputGo, OutputGo: TD × {up, down} → Boolean
      Modules        inputGate, outputGate: sluice-gate
                     SensorSet: (low, middle, high) → sensor
      Connections        {(InputGo InputGate.go)
                          (OutputGo OutputGate.Go)}
      Vars s1,s2: (low, middle, high)
      Axioms
```

-- a fault occurs whenever the sensors give conflicting outputs --

Fault: SensorFault $\leftrightarrow$ ($\exists$s1 SensorSet(s1).above $\wedge$ $\exists$s2 s2<s1 $\wedge$ $\neg$SensorSet(s2).above)

end tank

The graphic representation of the class tank is depicted in fig. 6, while its tree of parts is in fig. 7.

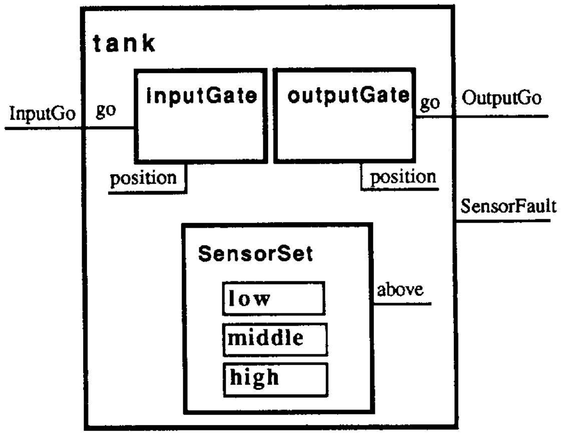

Figure 6. The picture of class tank.

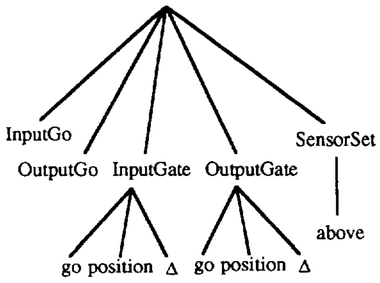

Figure 7. The tree of parts of class tank.

The following closed formula is obtained as result of the translation of the above TRIO+ specification into TRIO:

Always((InputGo $\leftrightarrow$ InputGate_go) $\wedge$
(OutputGo $\leftrightarrow$ OutputGate_go) $\wedge$
(SensorFault $\leftrightarrow$ ($\exists$s1 SensorSet(s1).above $\wedge$ $\exists$s2 s2<s1 $\wedge$ $\neg$SensorSet(s2).above)) $\wedge$
(OutputGate_position=up $\wedge$ OutputGate_go(down) $\rightarrow$
Lasts (OutputGate_position=mvdown, OutputGate_Δ) $\wedge$
Futr (OutputGate_position=down , OutputGate_Δ)) $\wedge$
(InputGate_position=up $\wedge$ InputGate_go(down) $\rightarrow$
Lasts (InputGate_position=mvdown, InputGate_Δ) $\wedge$
Futr (InputGate_position=down , InputGate_Δ)) $\wedge$
$\forall$s1(Becomes(SensorSet_above(s1)) $\rightarrow$ Lasts(SensorSet_above(s1),3) $\wedge$
Lasted($\neg$SensorSet_above(s1),3)))

Notice that, in the TRIO names deriving from the concatenation of names of TRIO+ class parts, an underscore character "_" separates the components of the identifier; any possible dot notation is cleared: InputGate.go becomes InputGate_go. It can also be noted how, for the presence of the array SensorSet of components of class sensor, the zero-ary predicate *above* becomes a unary predicate on the domain (low, middle, high); a universal quantification on this domain is added to all the axioms of class Sensor.

4 Conclusions and future work

We presented TRIO$^+$, an object oriented logic language for modular system specification, which allows the specifier to structure the description of the system in distinct, separate and reusable modules. TRIO$^+$

was used successfully in the specification of hardware and software systems of significant architectural complexity, like pondage power stations of ENEL, the Italian electric energy board. Systems of this kind are highly structured and exhibit quite a complex behavior: they are governed by management programs whose validity lasts several days or weeks, respond with flexible and adaptable actions to a large variety of events coming from the surrounding environment, and include components with intrinsic time constants ranging from several hours (for a water basin) to microseconds (for the electronic circuitry that controls the power distribution).

The experience gained in this activity confirmed that for a specification language the possibility of structuring specifications, that is to divide them into parts and to define suitable abstraction levels to hide unessential details, is as crucial as for a design language. In particular, for a logical language, as the number and length of formulas increases beyond a certain threshold (which for humans is unfortunately quite low) then a significant or even prevalent part of the specifier time and effort is spent just in purely syntactical activities, like checking the name and type of entities, or the consistency between use and definition of an object.

In such a framework it was noted that a graphic notation, with its ability to convey a great deal of information in a compact, structured and intuitively appealing form, can be of great help. Also, the availability of language-dependent tools, such as syntax-directed editors, graphic editors, automatic consistency checkers, provides a support to exactly those parts that are not conceptually relevant nor difficult, but become painfully intricate and time-consuming when the specification increases in size. The use of such automatic tools allows the specifier to concentrate his efforts on the conceptually relevant and challenging aspects of the modelled system. We also point out that, unlike most informal specification languages and methods, which provide a graphic notation without an associated formal and rigorous semantics, TRIO$^+$ combines in a suitable linguistic frame the possibility to structure the specification into modules and the description of semantic aspects, expecially those regarding the temporal behavior.

The support to semantic activities on the specification, like temporal analysis by means of testing, simulation, and property proof, is made possible by the fact that TRIO$^+$ can be readily translated into TRIO, a formal language for which methods and tools to support executability are available. The execution of the TRIO$^+$ specification of a system is certainly possible with the same computational effort necessary to execute the TRIO specification of the same system. In fact, the formulas obtained from the translation of TRIO$^+$ are not longer than those one would write when constructing the TRIO specification from scratch. Thus, the use of TRIO$^+$ as a methodology to structure TRIO specifications does not impose any overhead for what regards the size of the specification or its execution. In practice, there should even be an advantage in executing TRIO$^+$ specifications instead of semantically equivalent TRIO$^+$ formulas, since structuring the system into modules allows one to partition it into independent parts that can be executed separately, at a substantially lower cost than the execution of the system in its entirety.

Future work will thus be devoted to the construction of tools that provide a syntactic support to the construction and manipulation of large specifications, and a semantic support to the validation activities, taking advantage of the structure of the specification to increase the algorithms efficiency and to improve the quality of the visualization and presentation of the system during the simulation phases.

As noted above, such activities are possible because TRIO$^+$ can be easily translated into TRIO. This is a clear symptom of the fact that TRIO$^+$ is not substantially more expressive than TRIO. We can note that this happens because TRIO$^+$ classes define macro constructs that can be used in the description of complex system, but do not define truly new mathematical entities. The main reason for this resides in the fact that TRIO$^+$ axioms, which ultimately convey the meaning to TRIO$^+$ specifications, are written in term of the *items* only, which are the elementary parts of a first order logic. Thus, the axioms can be expressed in terms of TRIO. It is not possible, in the language presented here, to declare variables representing values (i.e. objects) of a TRIO$^+$ class, define on them higher-order operators or perform quantifications. This might seem a limit of the language, but was a deliberate choice intended to preserve the pleasant features of TRIO regarding execution of specifications.

A complementary approach is however possible, whereby one denotes explicitly sets of objects of a class and defines their properties using a higher order logic, thus augmenting considerably the expressiveness of the notation. On the other hand, the adoption of a higher-order logic would make any form of execution of the specifications virtually unfeasible, although still possible in principle, by means of translation of higher-order formulas into first order form, through well known techniques [End 72]. In this case the complexity of the tableaux-based algorithms would increase of several orders of magnitude.

The definition and use of a language of this kind would involve typical considerations of trade-offs among expressiveness, generality, and flexibility on one hand, simplicity and efficiency of representation and execution on the other. A draft version of TRIO$^+$ that includes such features has been defined in [M&S 90], but its complete formal semantic is still to be provided. Such a semantics cannot be "transformational", i.e. obtained by syntactic translation as in section 3.5, because the language is much more powerful than TRIO; thus, the semantics of a class must be directly assigned via suitable, more complex interpretation structures.

The same direct approach can also be used to give an alternative, but equivalent, version for the semantics of section 3.5 for the language presented in this paper. For a simple class, this can be done in a very straightforward way by an usual TRIO structure adequate for the logical conjunction of the axioms of the class. For a complex class, a compositional definition is possible, which combines in a suitable way the structures of its components to build one structure for the class. Such a definition has not yet been developed, because our interests were mainly devoted to maintain compatibility with TRIO, in order to reuse existing tools, algorithms and results, but it will be part of our future work.

References

[BPM 83] M. Ben-Ari, A. Pnueli, and Z. Manna, "The Temporal Logic of Branching Time", Acta Informatica 20, 1983.

[CHJ 86] B. Cohen, W.T. Harwood, M.I. Jackson, "The Specification of Complex Systems", Addison Wesley Publ. Comp., Reading MA, 1986.

[D&K 76] F. DeRemer, H. Kron, "Programmaing-in-the-Large Versus Programming-in-the-Small", IEEE Transactions on Software Engineering, SE-2, (June 1976):80-86.

[DeM 78] Tom De Marco, "Structured analysis and system design", Yourdon Press, New York, NY, 1978.

[End 72] H.B. Enderton, "A mathematical introduction to logic", Academic Press, London, 1972.

[F&M 91] M.Felder, A.Morzenti, "Real-Time System Validation by Model-Checking in TRIO", 1991 Euromicro Workshop on Real-Time, Paris, 1991.

[GMM 90] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO, a logic language for executable specifications of real-time systems", The Journal of Systems and Software, Vol. 12, No. 2, May 1990.

[M&S 90a] A. Morzenti, P. San Pietro, "TRIO$^+$ an Object Oriented Logic Specification Language", ENEL-CRA Research Report, January 1990 (in Italian).

[M&S 90b] A. Morzenti, P. San Pietro, "An Object-Oriented Logic Language for Modular System Specification", Int. Report no. 90.027, Politecnico di Milano, Dipartimento di Elettronica, 1990.

[MBM 89] A. Mili, N. Boudriga, F. Mili, "Towards structured specifying: theory, practice, applications", Ellis Horwood Ltd., Chichester, England, 1989.

[Mey 86] B. Meyer, Genericity versus Inheritance, OOPSLA, Portland, Oregon, 1986

[Mey 88] B. Meyer, "Object-oriented Software Construction", Prentice-Hall, 1988

[MMG 90] A. Morzenti, D. Mandrioli, C. Ghezzi, "A Model Parametric Real-Time Logic", Int. Report no. 90.010, Politecnico di Milano, Dipartimento di Elettronica, 1990.

[Mor 89] Angelo Morzenti, The specification of real–time systems: proposal of a logical formalism, PhD Thesis, Dipartimento di Elettronica, Politecnico di Milano, 1989.

[MRR 89] A. Morzenti, E. Ratto, M. Roncato, L. Zoccolante, "TRIO, a Logic Formalism for the Specification of Real-Time Systems", IEEE Euromicro Wirkshop on Real-Time, Como, Italy, 1989.

[R&U 71] N. Rescher and A. Urquhart, "Temporal Logic", Springer Verlag, Vienna-New York, 1971

[Smu 68] Raymond M. Smullian, "First order Logic", Springer Verlag, 1968.

[War 86] Paul T. Ward, The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, IEEE TSE, Vol. SE–12, no. 2, Feb. 1986.

[Weg 88] P. Wegner, Object-oriented concept hierarchies, Brown University, Technical Report, 1988

[Wir 77] N. Wirth, Towards a Discipline in Real-Time Programming, Comm. ACM 20-8, 577-583, Aug. 1977.

[Wol 83] P. Wolper, "Temporal logic can be more expressive", Information and Control 56, 1983.

[Y&C 79] E. Yourdon e L. L. Constantine, Structured design, Prentice Hall, Englewood Cliffs, NJ, 1979.