From Formal Models to Formally Based Methods: An Industrial Experience

EMANUELE CIAPESSONI
ENEL-SRI PEA-NTE
ALBERTO COEN-PORISINI
Politecnico di Milano
ERNANI CRIVELLI
ENEL-SRI PEA-NTE
DINO MANDRIOLI
Politecnico di Milano
PIERGIORGIO MIRANDOLA
ENEL-SRI PEA-NTE
and
ANGELO MORZENTI
Politecnico di Milano

We address the problem of increasing the impact of formal methods in the practice of industrial computer applications. We summarize the reasons why formal methods so far did not gain widespread use within the industrial environment despite several promising experiences. We suggest an evolutionary rather than revolutionary attitude in the introduction of formal methods in the practice of industrial applications, and we report on our long-standing experience which involves an academic institution, Politecnico di Milano, two main industrial partners, ENEL and CISE, and occasionally a few other industries. Our approach aims at augmenting an existing and fairly deeply rooted informal industrial methodology with our original formalism, the logic specification language TRIO. On the basis of the experiences we gained we argue that our incremental attitude toward the introduction of formal methods within the industry could be effective largely independently from the chosen formalism.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies* (e.g., object-oriented, structure); D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Design, Documentation, Verification

Additional Key Words and Phrases: Formal models, industrial applications, object orientation, specification, supervision and control, technology transfer

Authors' addresses: E. Ciapessoni, E. Crivelli, P. Mirandola, ENEL-SRI PEA-NTE, Via Volta, 1, Cologno Monzese, Milano, Italy; email: {ciapessoni; crivelli; mirandola}@pea.enel.it; A. Coen-Porisini, D. Mandrioli, A. Morzenti, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, Milano, I-20133, Italy; email: {coen; mandriol; morzenti}@elet.polimi.it

1. INTRODUCTION

The usefulness of formal methods (FMs) in the development of industrial computer applications is one of the most controversial issues and a source of debates within the computing community [Bowen and Hinchey 1995a; Gerhart et al. 1994; Hall 1990; Pfleeger and Hatton 1997]. A considerable number of successful experiences have been reported on the application of FMs to real industrial projects [Alspaugh et al. 1992; Bowen and Hinchey 1995a; Faulk et al. 1994; Hall 1996; Lagnier et al. 1995].[1] Almost unanimously they confirm the major expected benefits, mostly in terms of early error detection and, ultimately, in terms of increased reliability. Nevertheless their adoption within standard industrial practices is still an exception, and several typical criticisms and skepticisms are emphatically claimed by their opponents. Most of such criticisms are what Hall [1990] and Bowen and Hinchey [1995a] identified as the "myths of formal methods." Among them the most common are

—FMs require too much mathematical skill;

—FMs increase development costs;

—FMs do not guarantee error freedom (they have often been "oversold");[2]

—FMs are not well supported by suitable tools and technology transfer. Even when they are successful, success is the result of the skilfullness of few talented people and cannot be reproduced in routine development processes.

A recent nice synthesis of most of the above claims has been provided by the following folk assertion attributed to John Rushby: "The problem with FMs is that they are just *formal, not methods*." In fact, having a good formalism to specify, analyze, implement, and verify computer applications, even if well supported by sophisticated tools, is not enough if their users are not taught and trained to apply it to their practical problems with suitable guidelines.

A further confirmation of this analysis can be obtained from Hinchey and Bowen [1995] which contains a nice collection of useful case studies: in most cases they report on successful application of FMs to industrial projects. We mention in particular Fitzgerald et al. [1995, chap. 14] where the use of a formal method is favorably compared against an informal one. Nevertheless, Craigen et al. [1995, chap. 17] point out a number of weaknesses which still hinder adoption of FMs within the industrial environment: to gain better acceptance the authors recommend to devote more efforts to training and technology transfer; to provide more convinc-

[1]See also the Formal Methods Application Database, a database containing descriptions of industrial applications of formal methods, developed in the context of the project FMEInfRes (Formal Methods Europe Informazion Resource); available via http://www.cs.tcd.ie/FME/.
[2]Hall's first myth was "FMs can guarantee that software is perfect." The realization that this claim was clearly unrealistic resulted in the opposite criticism.

ing evidence of FMs cost effectiveness through rigorous measurements; and to guarantee compatibility with other useful software engineering approaches.

In this article, we try to contribute to achieve such a goal by reporting on our (10 years long) experience in introducing in a standard and systematic way the use of a formal development method within an industrial environment. This has been from the very beginning a joint research effort of an industrial and an academic group operating in the field of supervision and control systems (SCS) (plant control, energy management, traffic control, etc.). This application field is generally considered as a natural domain for exploiting FMs, since SCS usually are safety-critical systems, highly demanding in terms of reliability, generality, and reusability requirements, and involve major investments.

At the early stages of our research we developed a formalism—the TRIO specification language—to specify, analyze, and verify SCS requirements [Ghezzi et al. 1990; Morzenti and San Pietro 1994]. The first experiences confirmed that TRIO fostered writing precise and well-understandable requirements that could be rigorously confronted against implementation. They also showed, however, that if we wanted to pursue a wide adoption of TRIO, it was necessary to complement the original formalism (and its tools) with carefully devised methods in order to drive their use. This led us to adopt a strategy whose corner stones are the following:

(1) *Augmenting, not destroying existing practices*: Industrial environments, mainly in large companies, have a strongest inertia: seldom drastic changes ("revolutions") are successful.[3] Therefore, we decided to pursue *incrementality*: rather than trying to impose people to change their methods, we tried to convince them to *add* some amount of formalization in few critical points, by asking for little, gradual, learning. This also guaranteed compatibility with previous standards and document readability by people unfamiliar with formalisms.

(2) *Adopting an evolutionary approach to application development*: In principle, the TRIO formalism could support an approach that favors (a) building formal specifications from scratch analyzing the application needs and (b) their formulation in terms of formal and informal statements hand-in-hand. Were our industrial environment a small firm with recently hired young people, we would suggest to follow this approach. Instead, we adopted an evolutionary approach because we wanted to minimize "intrusion" in the existing practices (see Section 7 for more remarks on such "social aspects" of methodologies to exploit FMs).

[3]The history of programming languages is quite illuminating from this respect. Even the recent adoption of object-oriented (OO) approaches, which has been advertised as—and in some sense is indeed—a revolution, has gained acceptance by exploiting an evolutionary approach: C++ has been more successful than other "more radical" OO languages most likely thanks to its smooth evolution from C; the early OO analysis methods [Rumbaugh et al. 1991] were derived by adapting previous structured analysis ones.

The result of such an evolutionary attitude is a life-cycle model that could be named a "double spiral model," as it contains two nested iterations, in the requirement and in the architectural design phases. The model has many similarities with the "V" development model [STARTS 1987], which is widely applied in the area of embedded system design, and can be roughly described as follows.

(1) An informal document written in natural language prose is produced. It is a first formulation of system requirements following a fairly well established standard in the SCS field [CEI-IEC 1996].

(2) The original informal document is analyzed and a first, partial formalization is produced using the TRIO formalism. This phase often requires some revision of the original document, possibly by means of an interaction between experts of the application domain and experts of the formalism. The result is a new document which is a *superset of the original informal one*—most likely a revision—augmented with some reformulation of informal statements in the formal language.

(3) A sequence of refinement/verification steps is carried out to smoothly obtain the architectural design and final implementation and delivery from the original requirement formulation. Every refinement and consequent verification may obviously require some revision of previous documents.

This accounts for the "double spiral" name: analysis, feedback, and revision occur both within a single phase and between different phases. In particular, in SCS applications the borderline between requirement specification and architectural choices is not as sharp. For instance, the requirements of a new-generation energy management system may be driven by the wish of exploiting new distributed architectures such as CORBA [OMG 1995] which allows one to integrate originally separated applications into a new, more powerful one.

Finally, very important spiral rounds occur at the outer level during the operational life of the system, since SCS are usually long-lived and involve major economic investments. Thus, they must evolve through several "generations" of technology and requirements evolution. This accounts for the high value that is attributed to specifications *evolvability* and *generality*, i.e., the ability to adapt them with minor changes to the needs of different cases [Lutz 1997].

The goal of this article is twofold: on one hand we provide an overview of the strategy we adopted to introduce a formal development method into an already well established industrial development process. In particular we restrict our attention to the early phases of the process, discussing how TRIO can be used to complement the already existing informal specification method currently in use at ENEL. On the other hand, we focus on a few meaningful and critical issues that can illustrate at best the benefits obtainable by exploiting formalization, since dealing systematically with all topics addressed by an industrial specification method for SCS (motivating the needs for the described system, explaining the rationale of its essential features, classifying and categorizing its requirements, etc.) is far beyond the scope of this article. In particular, we discuss in some detail the approach we followed to cope with two aspects that are very relevant in the context of SCS such as fault tolerance and configurability by presenting several examples derived from industrial projects.

A last introductory remark regards the choice of centering the proposed method on TRIO: being the developers of this notation, we feel confident of its qualities and actual industrial applicability. However, all the provided methodological suggestions and guidelines apply as well to most of the available formalisms [Abrial 1993; Bolognesi and Brinksma 1987; Chandy and Misra 1988; Dürr and Plat 1995; Jones 1990; Ravn et al. 1993; Spivey 1992], especially those based on logical or equational notations, and including object-oriented constructs.

Our approach has some similarity with a few proposals to integrate formal with informal methods [Larsen et al. 1991; Mander and Polack 1995; Weber 1996] to achieve the benefits of both sides. Also similar is the attempt to enrich existing informal specification notations with a formally defined semantics [Fuggetta et al. 1993; Petersohn et al. 1994]: we share with such approaches the wish of adding some amount of formalism to existing practices based on informal or semiformal notations. In this article however, our attention is focused on supporting a specification *method* by exploiting some suitable formalism, not on *integrating* existing *notations*, whether formal or not.

The article is organized as follows. Section 2 summarizes the specification method used by ENEL to produce an informal specification document. Section 3 provides a short summary of the TRIO formalism so that in Section 4 we can explain our method to derive a TRIO formalization—better: *enrichment*—of the original informal document. The method is illustrated through a few running examples derived from real projects that have actually been completely developed. A further complementary case study is reported in Section 5. Section 6 provides a few hints on how the method affects the other phases of the life-cycle model. Finally, Section 7 completes the presentation of our experiences, integrates the presented method with a few lessons we learned on organizational aspects of the introduction of FMs in industrial environments, and offers some concluding remarks.

2. THE INFORMAL ENEL SPECIFICATION METHOD

The informal requirement specification method presently in use at ENEL is obtained, with minor changes, from the IEC standard [CEI-IEC 1996] which was jointly developed by several European industries in the field of SCS applications. As most requirement specification methods, it essentially consists of a *system analysis* which leads to a system *requirement specification document*. Since the focus of this article is on the smooth evolution from the informal specification method toward a more formal approach, the

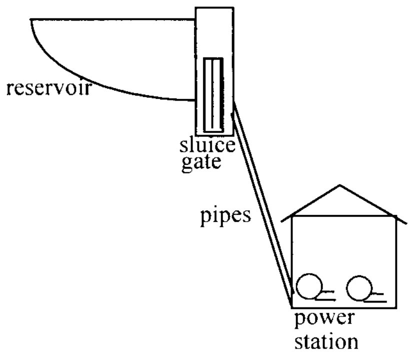

Fig. 1. A hydroelectric power plant.

initial ENEL methodology is, in some sense, an "independent variable," although some coordination between the specification *method* and the *language* used to document, and possibly formalize the requirements, is unavoidable.

SCS typically consist of some plant, physical process, or machine to be controlled along with some controlling devices. The system structure can be naturally described as a collection of objects, while SCS objectives (such as producing some amount of energy within some safety constraints, controlling the trajectory of a vehicle, etc.) can be described as tasks or functions to be performed. As a consequence, the ENEL method is, in some sense, both *object oriented and function oriented*. Thus, system analysis consists of building two different views: a *static view*, which defines the system structure, and a *dynamic view*, which defines the system behavioral properties.

The static view identifies the main components (the environment in which it operates, the process or plant to be controlled, etc.) and the relations occurring between them (e.g., physical and logical connections). For instance, in an oversimplified view, a hydroelectric power plant could be described as in Figure 1 where its main components (a reservoir, and a power station connected through pipes) are shown.

The dynamic view defines all the relevant system properties. In most cases such properties are expressed in an *operational style*, i.e., by means of *state variables* (e.g., temperatures, pressures, the level of a reservoir, the power produced by a plant, etc.) and *state transformations*, or *transitions* (e.g., the opening or closing of gates or valves, the switching of electric connections, etc.). Since the relevant properties of SCS are usually quite numerous and may differ deeply in scope and relevance, it is important to structure them according to a rational and well-understood categorization. ENEL's method recommends a fairly detailed categorization of system properties among which the most relevant ones are the following:

—*Functional properties*, related to data monitoring and elaboration: For instance, a typical requirement could be formulated as "the Acquisition and Preprocessing resource must periodically acquire from plant sensors the measurements of 31 analog variables (mostly pressures and temperatures) and 3 digital variables."

—*Performance properties*, expressing the temporal requirements of the system: Usually, they represent critical requirements, since most SCS belong to the class of real-time systems. A typical example is the time within which an alarm must be managed.

—*Safety properties*, including system protection from external disturbances, built-in safety mechanisms and protections against risks, suitable safety rules for the personnel.

—*Relevant external conditions* under which the system has to operate during its mission: Every system—or component thereof—is "plugged" into a context or environment: a whole plant is part of an energy system at country level and interacts with operators; a control subsystem receives inputs from sensors, etc. It is therefore useful to specify clearly the *assumptions* on the behavior of the context of a given module (system component) for two reasons: (a) the module under consideration will take these assumptions as *hypotheses* under which it must operate; (b) the same assumptions represent suitable requirements for the other parts of the system—including operators—that must be fulfilled to guarantee the global behavior of the system. For example, an alarm-managing module may assume that sensors—and, therefore, the controlled plant—do not generate more than 10 alarms per second: under this hypothesis its requirements state that all of them must be served within 10 seconds.

—*Definition of system behavior under permanent or transient faults*: In some cases, assumptions about the external conditions can be usefully categorized into *normal conditions* and *exceptional*—or even *faulty*—conditions. For instance, one could state that a faulty plant could generate up to 20 alarms per second. In this case the alarm manager module must be able to serve at least 10 of them within 13 seconds and the remaining ones within 20 seconds. Thus, the whole specification could have the following shape:
- under normal conditions, the module must guarantee some given requirements;
- under exceptional conditions it must instead guarantee some other requirements.

The above properties are grouped into a larger category, named *operating properties*. There are also other types of properties specifying managing and maintenance procedures, services supplied by vendors and users, documentation requirements, training procedures, HW and SW compatibility, communications, etc.

Among these other properties special attention is devoted to *configurability requirements*, since system configurability is a fundamental quality for most industrial systems. Designing a configurable system is a typical example of the *design-for-change* principle stated by Parnas. In fact, in most cases a system is a member of a family of similar but not identical elements. Therefore, it is fundamental to analyze precisely the parameters that are likely to change when moving from one instance of the system family to another. For instance, in the case of hydroelectric power systems many similar plants may differ in the number, type, and altitude of reservoirs, in the number and power of power stations, etc.

As we said the result of system analysis is a specification document which is structured into several chapters and sections reflecting the above categorization. Notice that the initial part of this document is usually devoted to motivate the needs for the system, explain its main goals or *mission*, and provide some background information suitable to explain the rationale of its specification and to guide its realization.

3. A SHORT SUMMARY OF THE TRIO LANGUAGE

TRIO includes a basic logic language for specifying, in-the-small, an object-oriented extension, which supports writing modular reusable specifications of complex systems, and a further extension which includes predefined higher-level, application-oriented notions such as events, states, processes, pre- and postconditions.

3.1 The Basic Logic Language

TRIO is a first-order temporal logic language that supports a linear notion of time: the *Time Domain* is a numeric set equipped with a total-order relation and the usual arithmetic relations and operators (it can be the set of integer, rational, or real numbers, or any interval thereof).

TRIO formulas are constructed in the classical inductive way, starting from terms and atomic formulas. Besides the usual propositional operators and the quantifiers, one may compose TRIO formulas by using a single basic modal operator, called *Dist*, that relates the *current time*, which is left implicit in the formula, to another time instant: the formula $Dist(F, t)$, where F is a formula and t a term indicating a time distance, specifies that F holds at a time instant at t time units from the current instant.

For convenience, TRIO items (variables, predicates, and functions) are distinguished into time-independent (TI) ones, i.e., those whose value does not change during system evolution (e.g., the altitude of a reservoir) and time-dependent (TD) ones, i.e., those whose value may change during system evolution (e.g., the water level inside a reservoir)

Several derived temporal operators can be defined from the basic *Dist* operator through propositional composition and first-order quantification on variables representing a time distance. A sample list of such operators is given in Table I.

Table I. A Sample of TRIO-Derived Temporal Operators

Operator	Definition	Explanation
$Futr(A, d)$	$d > 0 \wedge Dist(A, d)$	A holds d time units in the future
$Past(A, d)$	$d > 0 \wedge Dist(A, -d)$	A holds d time units in the past
$Alw(A)$	$\forall d Dist(A, d)$	A always holds
$Som(A)$	$\exists d Dist(A, d)$	Sometimes A holds
$Lasts(A, d)$	$\forall d'(0 < d' < d \rightarrow Dist(A, d'))$	A will hold over a period of length d
$Lasted(A, d)$	$\forall d'(0 < d' < d \rightarrow Past(A, d'))$	A held over a period of length d in the past
$WithinF(A, d)$	$\exists t(0 < t < d \wedge Dist(A, t))$	A will happen within d time units in the future

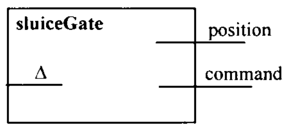

Fig. 2. The graphical representation of class sluiceGate.

3.2 The Object-Oriented Language Extension

The specification of large and complex systems requires the availability of constructs supporting modularization, abstraction, and reuse.

To this purpose TRIO has been enriched with concepts and constructs from object orientation. Among the most important added features are the ability to partition the universe of objects into classes, inheritance relations among classes, and other mechanisms such as genericity to support the reuse of specification modules and their incremental development (for the sake of brevity we do not illustrate, in the following, genericity and inheritance, which have been defined in a fairly standard way: the interested reader is referred to Morzenti and San Pietro [1994]).

TRIO is also endowed with a graphic representation of classes in terms of boxes, lines, and connections to depict class instances and their components, information exchange, and logical equivalence among (parts of) objects.

Classes denote collections of objects that satisfy a set of axioms. They can be either *simple* or *structured*—the latter term denoting classes obtained by composing simpler ones. A simple class is defined through a set of axioms premised by a declaration of all items that are referred therein. Some of such items are in the *interface* of the class, i.e., they may be referenced from outside it in the context of a complex class that includes a module that belongs to that class. Figure 2 displays the graphical representation of a simple class specifying a sluice gate in a pondage power plant: visible items are the *position* of the gate and the *command* to govern it, while the time constant Δ (the time necessary to open or close it) is hidden.

The corresponding textual definition of the class *sluiceGate* is reported below.

```
class sluiceGate
visible command, position
   temporal domain integer
   TD Items
      predicates command({up, down})
      values position: {up, down, mvup, mvdown}
   TI Items
      const Δ: integer
   axioms
      vars t: integer
      go_down: position=up ∧ command(down)
         → Lasts(position=mvdown, Δ)
/* This axiom states that if the gate position is up and command down
is given, then the gate will be moving in the down direction for Δ time
units and then it will be in the down position */
. . .
/* Similar axioms specify the behavior of the gate in the up position
and in the case a command occurs when the gate is already moving */
end sluiceGate
```

Figure 3 shows the graphical representation of a sample structured class: a reservoir consists of several *modules*: *inputGate* and *outputGate,* which are instances of the class *sluiceGate*; two actuators, each one to control a sluice gate, and a transducer which measures the water level in the reservoir. The external plant is able to send four commands to control the reservoir, to open or close each sluice gate. *Connections* may relate items in the scope of a class definition, to state in a visual fashion their equivalence. For instance, in Figure 3 the item labeled *command* connecting *actuator1* with *inputGate* specifies that the same item is visible, and therefore can be used by both classes within the *reservoir* class definition. Notice also that TRIO allows one to connect items with different identifiers (provided their types are compatible): for instance the *open* item of actuator1 is named *openInput* in *reservoir* and is therefore visible with that identifier outside *reservoir*. This feature is useful when a system is built by aggregating existing specifications in a bottom-up way, possibly reusing previously defined specifications.

3.3 Ontological Extensions

On the basis of the first experiences in the application of TRIO to real-life industrial projects, the language was further enriched by means of so-called *ontological constructs*, which support the natural tendency to describe SCS (as well as other kinds of systems) in a more operational way, i.e., in terms of states, transitions, events, processes, etc.

An *event* is a particular predicate that is supposed to model instantaneous conditions such as a change of state or the occurrence of an external stimulus. Events can be associated with *conditions* that are related causally or temporally with them. From the causal viewpoint, a condition for an

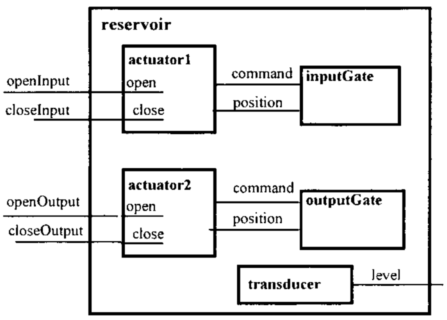

Fig. 3. Graphic representation of the structured class reservoir.

event can be *necessary* or *sufficient*; from the temporal viewpoint, conditions are divided into *preconditions* that refer to the time instants preceding the event, and *postconditions*, that refer to the instants following it.

A *state* is a predicate representing a property of a system. A state may have a duration over a time interval; changes of state may be associated with suitable predefined events and conditions. Altogether, events, states, and conditions define a comprehensive model of the system evolution.

For instance, the class definition given below provides a partial reformulation of the *sluiceGate* class defined in Section 3.2 which exploits ontological constructs.

```
class sluiceGate
visible command, position
temporal domain integer
   STATE Items position: {up, down, mvup, mvdown}
   EVENT Items beginGo ({up, down}), endGo ({up, down}),
   command({up, down})
   TI Items
const Δ: integer
axioms
      vars t: integer
event beginGo (down)
pre_nec_suff position=up ∧ command(down)
post_nec Lasts(position=mvdown, Δ) ∧
            Futr(position=down, Δ)
      . . .
end sluiceGate
```

4. MAPPING INFORMAL SPECIFICATIONS INTO A FORMAL DOCUMENT

As stated in the introduction, a specification method could start the requirements specification using TRIO from the very initial phases. However, at the present state of the art, our method envisages to derive a TRIO document from an original informal document written along the lines described in Section 2. Thus, the TRIO document does not replace but

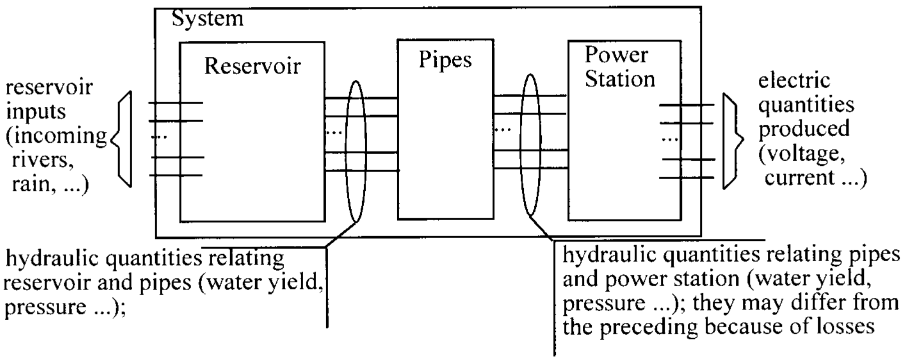
System
Reservoir
Pipes
Power Station
reservoir inputs (incoming rivers, rain, ...)
electric quantities produced (voltage, current ...)
hydraulic quantities relating reservoir and pipes (water yield, pressure ...);
hydraulic quantities relating pipes and power station (water yield, pressure ...); they may differ from the preceding because of losses

Fig. 4. The TRIO class structure of a pondage power station.

extends the original one so that the complete specification document remains readable by any reader by simply skipping mathematical formulas.

This section describes how this formalization step can be accomplished: Section 4.1 shows how the system structure can be described in terms of a class structure; then, in Section 4.2, we face the general problem of documenting system requirements as a suitable collection of formulas embedded within class definitions, while Sections 4.3 and 4.4 focus on the problem of formalizing fault tolerance requirements and system configurability, respectively. We ponder more deeply these topics because of their relevance for industrial applications and because our experience showed that major benefits were obtained here by moving from an informal—and often generic—requirement formulation to a careful and systematic formal analysis and documentation.

4.1 The Class/Object Structure of Specifications

The first step of the formalization process consists of writing a collection of boxes that correspond one-to-one to system's main components. For instance, Figure 4 displays the structure of a pondage power station as outlined in Figure 1.

Strictly speaking, TRIO is more class oriented than object oriented, since a specification consists of a collection of class rather than object definitions. This language feature matches our methodological needs, since in most cases we are not interested in specifying the properties of a single object: we need to talk about the level of any reservoir within a given class rather than the level of a particular reservoir. Only when configuring a particular system we describe the specific features of subclasses possibly down to single plants.

Then, the description of the system structure proceeds with the definition of the items composing each class. Most traditional specification methodologies—including ENEL's—are state oriented; that is they favor a system description in terms of state components (a reservoir level, a gate being open or closed, etc.) and of their evolution through state transformations.

Thus, the formalization activity consists of mapping such elements into suitable TRIO items. For instance, a reservoir level can be formalized, by exploiting ontological constructs, as a state, while the opening or closing of a gate can be formalized as an event, etc.

The description of the system structure is then completed by specifying the *connections* between classes. For instance, with reference to Figure 4, items such as water yield and pressure may be shared by *Pipes* and *Power station*.

Our method provides some suggestions that can help unexperienced users in carrying out this first step of the formalization activity:

—TRIO does not have an explicit notion of input/output relations among modules. Thus, for instance, an object "monitor" receiving an alarm from the environment and reacting by providing a shutdown command to the plant can be formally expressed by an item *alarm* and an item *shutdown* in the box *monitor*. However, the user can augment lines denoting items by giving them a direction (denoted by an arrowhead). This information can be exploited in later phases such as implementation and verification [Mandrioli et al. 1995], but it is not further considered in this article. Further guidelines can be borrowed from other methods such as Parnas and Madey [1995].

—The *connections* between TRIO classes must not be confused with possible *physical links* between the objects denoted by those classes: a TRIO connection simply denotes that the two items represented by the line segment (a level measure, a signal, etc.) are the same thing. Therefore, a physical link between objects, such as a pipe connecting a reservoir with a power station or an electrical junction between circuit components, does not necessarily result in a TRIO connection in the specification document. For instance, if we represent two short-circuited pins of two integrated circuits by a TRIO connection between the two items *voltage* belonging to the two circuits, the connection denotes that the two voltages are the same quantity. In other words the TRIO connection models the electric consequence of the physical connection between the two devices. Instead, in order to represent the physical connection occurring between a power station and a transformer cabin we need an explicit class "transmission line," since we have to take into account the loss of voltage due to the resistance of the line. Thus, in general denoting the fact the two objects are physically connected may be represented by a—possibly even TD—relation between the instances of the two classes.[5]

[5]In some complex cases we even recommend to encapsulate all facts describing physical connections among system components into a suitable class "topology": for instance in an energy distribution system we might include in such a class the description of the dynamic state of the distribution network where it is stated through suitable TD predicates whether a given station X is connected with another station Y.

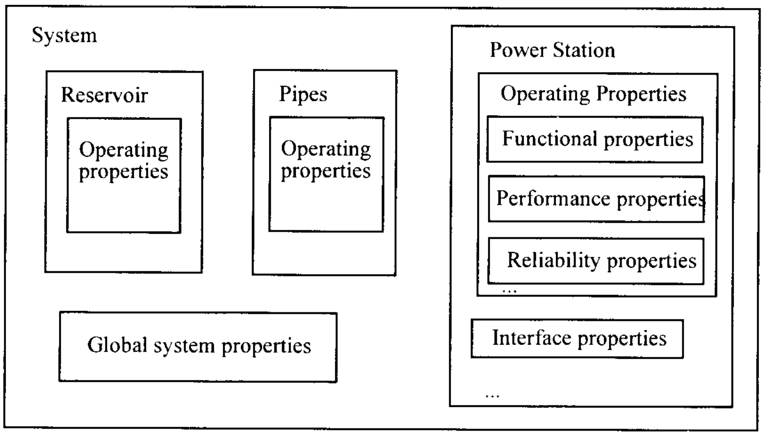

Fig. 5. Global class structure of the TRIO specification of the system introduced in Figure 1.

4.2 Formalizing System Properties

Once we have formalized the system structure by means of a suitable collection of TRIO classes, items, and connections we move to the formalization of the system properties.

Notice that also this activity follows the structure of the original informal document. Thus, as a first step, we group the (usually numerous) properties into several modules and submodules following the structure suggested in Section 2.

This is accomplished by using the main TRIO encapsulation mechanism, i.e., classes: as a result for every system component we have a class definition for each group of properties (unless they are particularly simple and few). The class mechanism is now used in a different way than when defining the system structure. In fact, in the previous case we used classes to specify the different components of a given system, whereas now we partition the different properties of the same system, or component thereof, into several classes. Therefore, in the most general and complex case, we come up with a general TRIO structure such as the one depicted in Figure 5, which reflects the object—and function—oriented nature of the ENEL method.

Notice that it is not necessary that every system component specifies every type of properties: for instance, interface properties make sense only for those components that involve some human-machine interaction.

Eventually, we move from "specification in-the-large" (i.e., system structure and properties) to "specification in-the-small," where we formalize single properties using pure TRIO formulas. To simplify at best this critical and often difficult activity, we used the following guidelines:

—*Not everything must be formalized*: The specifier must manage under his or her responsibility the trade-offs between the benefits in terms of precision, use of automatic tools, etc., and the costs in term of effort which in turn depends heavily on his or her experience and attitude toward managing the mathematical notation of the adopted formalism. In practical cases, we state in pure TRIO only a part of the system properties, whereas many of them would remain as prose text suitably inserted into the global structure specified above.

—*Use an operational attitude*: In most cases an operational attitude can help specifying the system behavior (mostly for the functional part), since it is common practice to describe system evolution through states and transitions, possibly triggered by external events. Similarly to other specification languages [Bolognesi and Brinksma 1987; Heitmayer and Lynch 1996; Ravn et al. 1993] TRIO supports such a specification style—without imposing it—through the constructs of pre- and postconditions, events, and processes (Section 3.3 showed a simple example of such a style of specification). Notice that this style of formalization is favored when a similar style is adopted in the informal version of the document.

—*Introduce some "thumb-rules"*: In order to help users who are not familiar with the formal notation, the language manuals should include a collection of "thumb-rules" to move from prose description to formal specification, pointing out a few typical traps and suggesting how to avoid them. A typical suggestion is to translate an informal "such that" into an implication when it is paired with a universal quantification and into a conjunction when it is paired with an existential quantification. For example, the sentence "a reservoir located in the Alps must not have an area greater than k m^2" becomes $\forall \mathrm{r}(\mathrm{alpine}(\mathrm{r}) \rightarrow \mathrm{area}(\mathrm{r}) \leq \mathrm{k})$ while the sentence "there are reservoirs in the Alps that are located at an altitude greater than h" becomes $\exists \mathrm{r}(\mathrm{alpine}(\mathrm{r}) \wedge \mathrm{altitude}(\mathrm{r}) > \mathrm{h})$.

Another typical suggestion is to avoid—whenever possible—the use of explicit quantifiers when specifying time-related properties: the TRIO notation includes a comprehensive and extensible set of derived temporal operators. These allow the user to substitute intricate formulas involving tricky applications of quantifiers with more abstract operators having intuitive and expressive names, thus fostering the association of formal requirements expression with their corresponding informal statement. For instance the formula Lasts(P, d) is more immediate and understandable than $\forall \mathrm{d}'(0 < \mathrm{d}' < \mathrm{d} \rightarrow \mathrm{Dist}(\mathrm{P}, \mathrm{d}'))$. Clearly, the usefulness and the style of such suggestions strongly depend on the cultural background of the industrial environment where the formal method must be adopted.

4.3 Dealing with Fault Tolerance Requirements

Industrially sized SCS are composed of several parts that interact in elaborate ways among them and with an—often adverse—environment, including aggressive physical, chemical, or electromagnetic agents. As a consequence the system behavior must comply to the system mission even when faults occur and fault tolerance requirements must be included among the customary automation system requirements, as illustrated in Section 2.

It is useful, however, from a methodological point of view, to distinguish requirements related with fault tolerance from requirements regarding system operation in absence of faults. Therefore, structuring the specification into parts describing "normal" functioning and other parts describing "exceptional" or "faulty" conditions can improve its readability and make the system easier to build and less subject to errors. Thus, functioning in the presence of faults can be described by means of variations (modifications or additions) of the normal operation specification. As it is often the case, the construction of specifications through incremental modifications and additions can be effectively managed by exploiting the inheritance mechanisms of object-oriented notations, as it is shown by the following example. A complementary technique is illustrated in Section 5.3.

Example 1. Suppose we want to specify a system subject to both hardware faults and disturbances which tend to change the value of a state variable, thus causing incorrect system operation.[4] Informally, the basic requirements on the system behavior are specified as follows:

(a) when a hardware fault occurs and remains unrepaired for at least *delta* time units, the system must be able to find the damaged part and put it off-line;

(b) the system operates normally only if the value of the state variable is not altered by a disturbance.

In order to formalize the above requirements which involve the distinction between normal and faulty operation, it is necessary to describe the presence of faults by modeling either the normal or the expected values as distinct items from the actual ones (in the present example items *actualValues* and *expectedValue* are modeled). These features are described in the class C whose structure is reported in Figure 6.

```
class C
... declarations corresponding to the graphical representation of
Figure 6
   axioms
        /* normFunct is the proposition that models the normal func-
        tioning of the system: it is defined as equivalent to the con-
        junction of $C_1$. . . $C_n$, the other axioms describing normal func-
        tioning in ordinary conditions */
```

[4]In SCS, faults are classified in a fairly specialized way and in most cases are either managed or detected by special-purpose hardware devices so that fault tolerance requirements can be stated without resorting to general-purpose fault models such as in Anderson and Lee [1981; 1990]; the present example, therefore, is intended only to show how fault tolerance requirements are included in the specification of a special class of ENEL's SCS; by no means it claims to illustrate a general model of fault tolerance requirements. On the other hand, both the specification method and the TRIO language can be easily exploited even within more general fault tolerance models such as Byzantine assumptions.

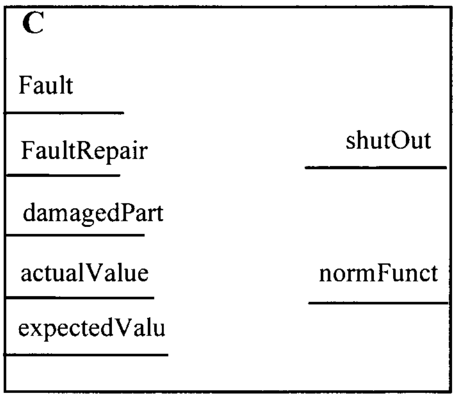

Fig. 6. Class C specifying basic system behavior.

```
        normFunct ↔ C1 ∧ . . . ∧ Cn
          /* the axiom below formalizes requirement a) */
        whenShutOut: shutOut(damagedPart) ↔
              Lasted(Since(¬FaultRepair, Fault), delta)
          /* the axiom below formalizes requirement b) */
        disturbFunct: normFunct → actualValue = expectedValue
    end C
```

Suppose now that we need to specify the following *additional* fault tolerance properties to make the system more robust in presence of faults:

(a) the damaged part should be put off-line only whenever the second unrepaired hardware fault occurs (it is assumed that the system can be managed correctly in presence of a single unrepaired fault);

(b) the system should operate normally if the value of the state variable, though recently altered for some time intervals, maintains the expected value for at least half of the time during the last *delta1* time units.

We specify these stronger fault tolerance requirements by defining an heir class of C, called *CFaultToler*, that inherits from class *C* and adds two modules, *HWFaulTol* and *DistFaulTor* that include the specification of properties of fault tolerance with respect to hardware faults and disturbances on the state variable, respectively (see Figure 7).

Let us first consider the property of tolerance to hardware faults. The module HWFaulTol is of class *clHWFaultTolerance*, defined by the following declaration and by Figure 8.

```
    class clHWFaultTolerance
    . . . item and module declarations according to the
    graphical representation of Figure 8 . . .
      axioms
      unrepairedFaults = FCount – RFCount
      criticalFailure ↔ unrepairedFaults ≥ 2
    end clHWFaultTolerance.
```

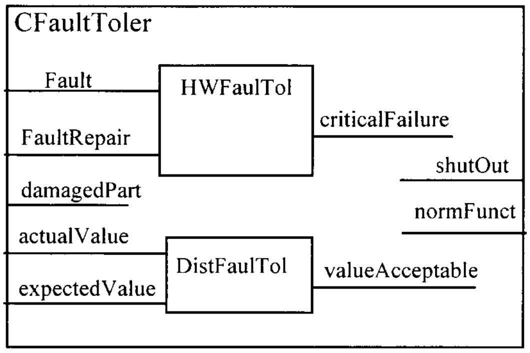

Fig. 7. The Class CFaultToler, heir of C.

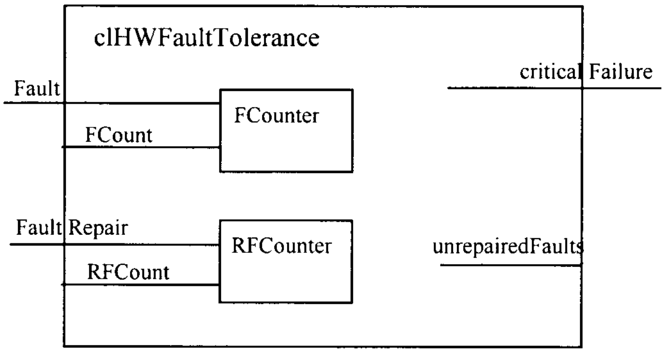

Fig. 8. Graphical display of class clHWFaultTolerance.

Class *clHWFaultTolerance*, in turn, includes two modules, *FCounter* and *RFCounter*, of class *EventCounter*, to model the counting operation of the events *Fault* and *FaultRepair*, respectively. A critical failure (i.e., one originated by some faults that cannot be "tolerated") occurs when the number of unrepaired faults (defined as the difference between the number of faults and the number of fault repairs) is greater than or equal to 2.

EventCounter is a typical example of *library class* definition: it includes axioms requiring that a generic predicate models an event, and specifying the counting operation. Its declaration is not reported here for the sake of brevity.

In a similar manner the module *distFaulTol* of class *clDisturbFaultTolerance* specifies the tolerance to disturbances on the value of the state variable (see Figure 9).

class clDisturbFaultTolerance

. . . module and item declarations as in the graphical representation of Figure 9.

axioms

measValue.condition $\leftrightarrow$ (actualValue = expectedValue)

valueAcceptable $\leftrightarrow$ measValue.heldOver(delta1) $\geq$ delta1 /2

end clDisturbFaultTolerance.

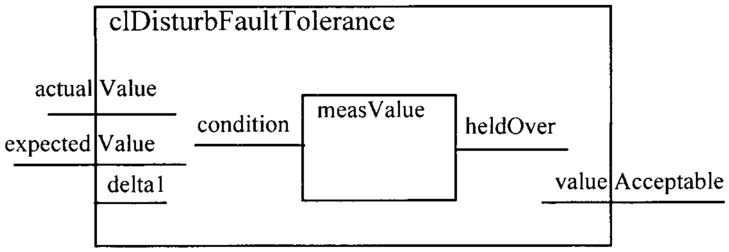

Fig. 9. Graphical display of class clDisturbFaultTolerance.

A module *measValue* of class *measure* is included in class *clDisturbFault-Tolerance* to specify that the actual value of the state variable has been equal to the expected value for at least half of the last *delta1* time units. Class *measure*—another typical example of library class—exports a function, *heldOver*, that for each possible length of the time interval ending at current time determines the total amount of time over which a given condition holds. The value of *heldOver* is simply the sum of the length of all intervals within the last period of time (whose length is denoted by the argument of *heldOver* itself), over which the condition held. Also the definition of class *measure* is not reported here.

Finally, class *CFaultToler* redefines[6] the axioms *whenShutOut* and *disturbFunct* describing the new conditions under which a hardware fault causes a shut out of the damaged part, and a disturbance of the value of the state variable hinders normal functioning. Notice that the axioms *whenShutOut* and *disturbFunct* of class *C* are redefined in *CFaultToler* in a similar way, by taking into account the items exported by modules *HW-FaulTol* and *DistiFaulTol* which specify the fault tolerance requirements.

```
class CFaultToler
inherit C [redefine whenShutOut, disturbFunct]
   . . . module declarations and connections according to the represen-
   tation of Figure 7
   axioms
         /* the damaged part is put off line iff a critical failure persists
         for at least delay time units*/
      whenShutOut: shutOut(damagedPart) ↔
                      Lasted(HWFaultTol.criticalFailure, delay)
         /* functioning is normal only if the comparison between the
         actual and the expected value gives an acceptable result*/
      disturbFunct: normFunct → DistFaultToler.valueAcceptable
end CFaultToler.
```

[6]Like many OO languages, TRIO supports both a strict, semantically monotonic notion of inheritance, when axioms in the heir class are only added to those of the ancestor, and a more liberal, nonmonotonic notion, when the axioms of the ancestor are redefined in the heir class.

4.4 Building Configurable Specifications of Configurable Systems

As pointed out in Section 2, industrial SCS are often replicated in several instances having a common structure but differing in the number and kind of components and/or parameters, to adjust operating conditions that may vary from plant to plant. For this reason, configurability criteria are carefully considered, starting from the very early phases of system analysis, with the same level of attention as the definition of aspects regarding the system's essential nature such as its mission and its tasks.

OO notations naturally support the definition of *families of classes,* since they embed concepts such as genericity and inheritance. Genericity helps configurability by giving the possibility to define classes that are parametric with respect to the kind of some components or to the value of some significant quantity. Inheritance helps factorizing universal requirements from special requirements. The use of such constructs results in a *static* form of configuration (i.e., the specification can be specialized to describe a given particular kind of system, whose distinctive features do not change at "run time").

A formal approach also helps in (automatically) designing a system, for instance by defining domains for configuration parameters, stating the extra requirements (e.g., fault tolerance), and then (automatically) checking the existence of a possible configuration that satisfies such extra requirements (formula satisfiability).

In what follows, we illustrate the use of TRIO constructs to specify and model the configurations of a complex artifact through a nontrivial example.

Example 2. Let us consider a pondage power plant that includes, in its simplest and most abstract form, the modules *reservoir* and *powerStation* representing its physical components. To deal explicitly with its possible configurations, we add a module *configuration* that includes all information on the plant configuration (see Figure 10). The configuration module does not correspond to a physical plant component: it is adopted as a systematic way to encapsulate the requirements concerning the allowed possibilities and the existing constraints on composing various kinds of plant components, thus determining the possible plant configurations.

In our example, a reservoir is characterized by the following features:

—*resDim*, the reservoir dimension: it can be large, medium, or small; adopting some simplification, we assume that the size of the reservoir is directly related to its maximum water delivery;

—*resAltid*, the altitude where the pondage is located.

—*mangPol*, the managing policy adopted for the reservoir: it can be *autonomous* (i.e., the water of the pondage is managed according to a

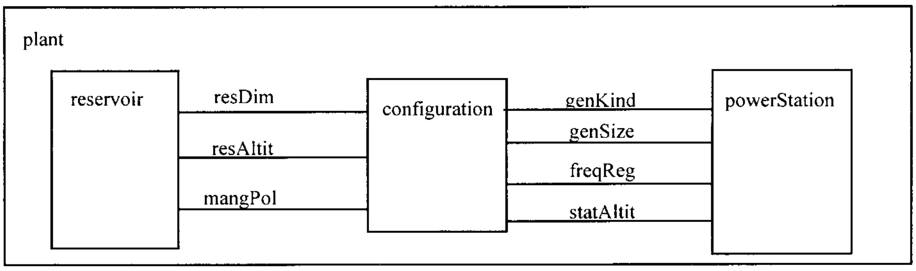

Fig. 10. The structure of a configurable plant.

production program defined autonomously by the plant) or *external* (i.e., the water is managed according to a production program that is defined outside the plant).

A station for power generation is characterized as follows:

—*genKind,* the type of the power-generating units in the station: there exist two kinds of unit: Francis and Pelton, which are respectively more suitable for low or high leap (difference in altitude) from the pondage to the station;

—*genSize*, the size of the generating units in the station, which depends on their nominal power (number of Volt Ampere); in pondage power plants one can have only small (generated power $<$ 10MVA) or medium size ($\geq$ 10MVA) power units;

—*freqReg*: the power units may (or may not) be able to finely adjust the frequency of the generated electric voltage around the nominal value of 50Hz;

—*statAltid*, the altitude of the station.

The class declaration for the module *configuration* includes the axioms describing how the plant can be configured by varying the *type* and *dimension* of its components.

```
type
typeResDim = (smallRes, mediumRes, largeRes);
   typeManPol = (autonomous, extern);
   genKindType = (Francis, Pelton);
   genSizeType = (small, medium);
class clConfig
visible . . .
   TI Items . . .
      const leap: natural; /*jump of the water flow from the reservoir
      to power station*/
            freqReg: boolean; /*true if the power units can regulate the
            frequency*/
   axioms
```

```
        /* the leap is defined as the difference in altitude between
        reservoir and power station */
    leap = resAltit – statAltit
        /* generating elements of the Francis kind must be used for
        small leaps (< 150 meters), Pelton ones for high leaps (> 250);
        for intermediate leaps both Francis or Pelton can be used but
        if the size of the reservoir (and hence its water delivery) is
        large Francis must be chosen */
    $(leap < 150 \vee (leap \le 250 \wedge resDim = largeRes)) \rightarrow$
                    $genKind = Francis$
    $leap > 250 \rightarrow genKind = Pelton$
    . . .
        /* Similar axioms state constraints among chosen generating
        units, reservoir size, managing policies, etc. For instance: "The
        size of the generating units is small if the leap is ≤ 250 and
        the reservoir is small or medium; it is medium otherwise
        . . . "*/
end clConfig.
```

Notice that the properties asserted in the *clConfig* class may imply more constraints on the configuration activity than those explicitly stated. We could derive the following, for instance:

(1) in a plant with a reservoir managed autonomously, Pelton power units, and frequency regulation, the leap must be high (greater than 250), and the reservoir size must be medium;

(2) no plant can exist with a median leap ($150 \le leap \le 250$), Pelton power units, and frequency regulation.

Often, different categories of plants can be defined on the basis of some choice of the configuration parameters. We can easily describe such categories by pairing the configuration class definition with the inheritance mechanism.

For instance, Figure 11 shows a simplified hierarchy of classes (defined as heirs of the class *plant* of Figure 10) corresponding to a variety of possible configurations. Each box representing a class contains the defining axiom and a few illustrative comments.

Notice that an heir class can be empty because it has properties that are incompatible with the constraints expressed in class *clConfig*. Thus, the properties of the configuration expressed in *clConfig* provide some discipline in determining the types and the dimensions of the components, by ruling out some unfeasible combinations.

The selection of the allowed possibilities from those deriving from a pure combination of the immediate heir classes of class *plant* can be done by the specifier through informal reasoning on the axioms of *clConfig*, as in points (1) and (2), or could be more effectively supported by semantic tools through the analysis of suitable formulas stating explicitly the feasibility of some combination of elements.

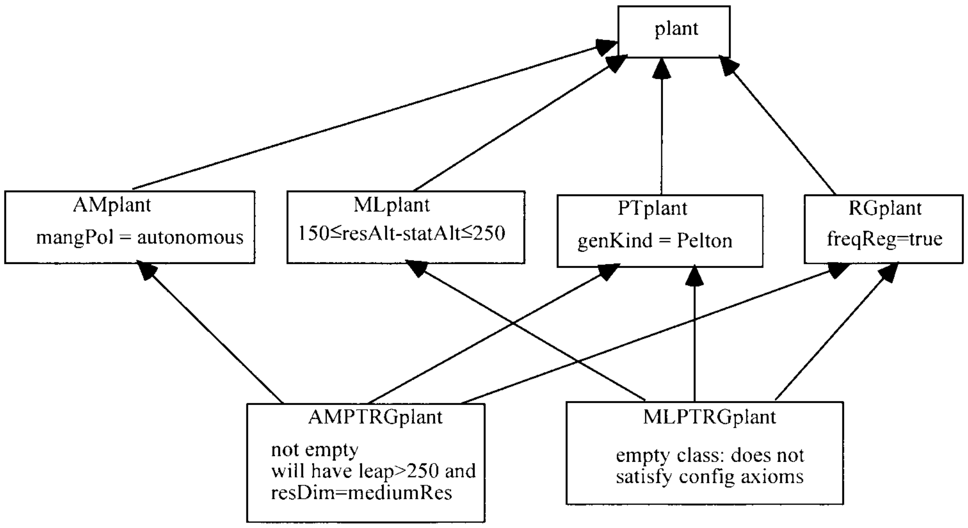

Fig. 11. Special categories of configured plants derived as heirs of the basic class.

In our example, class *clConfig* could include a further set of axioms expressing configurability of the plant, i.e., requiring that a certain combination of elements exists. These properties are expressed by axioms on variables representing the possible values of the configuration items. One of these axioms states the nonemptiness of class *AMPTRGplant*:

$$\exists mp \exists gk \exists fr (mp = managPol \wedge gk = genKind \wedge fr = freqReg \wedge$$
$$mp = autonomous \wedge gk = Pelton \wedge fr = true)$$

By analyzing the above axiom (possibly using an automatic tool), one can establish that it is consistent with the other axioms and that when *mp*, *gk*, and *rg* have the indicated values, the leap will be over 250 meters, and the reservoir size will be medium.

The method illustrated by the above example is recommended for static system configuration, i.e., a configuration which is set once and for all throughout system lifetime. Other, more dynamic forms of configuration take place when some features of the system can be changed during its operation; if these changes occur at a rate significantly slower than that of the usual operations, they are classified as *system reconfigurations*. In some cases the feature that can be changed is a single scalar parameter (formalized using a suitable TD item); other times it is a set: for instance, a monitoring system may periodically change the group of monitored values, or a control system may change the set of commands that can be issued. Section 5.4 illustrates how to deal with dynamic reconfiguration.

5. A CASE STUDY

In this section we present a case study centered on the specification of a real-life industrial application developed by ENEL in the framework of the OpenDreams (Open Distributed Reliable Environment, Architecture and Middleware for Supervision) ESPRIT project, as an example of a distributed supervision and control application.

The application concerns an information system designed to support ENEL's personnel in managing thermal power plant operations. The purpose of the system is to optimize power plant efficiency, to reduce operating and maintenance costs, and to avoid forced outages by implementing (separately or in combination) functions related to supervision, condition monitoring, performance monitoring, and fault diagnosis. The main problem with such a system is that each of the above functions is usually developed separately, as a *standalone* application. Therefore, when installed on the plant, each application needs its own field data acquisition system, data processing unit, data storage system, and man-machine interface. This in turn results in obvious drawbacks in terms of higher implementation, installation, maintenance, and training costs, increased operational complexity, confusion and distraction of the system users, and possibly even incorrect and unsafe plant management. To solve this problem, ENEL is trying to integrate the aforementioned functions in a unified environment, called the *Advanced Supervisory System*. The reported case study concerns the integration of a single *diagnostic* function (called PRECON) within the Advanced Supervisory System. To carry out such integration a TRIO specification of the whole Advanced Supervisory System was produced, including the description of the functions performed by PRECON and of those provided by the other components, along with their interactions.

In what follows, we describe some salient features of the specification to illustrate the methods introduced in the preceding sections. The interested reader can find the complete documentation in Benini et al. [1996]. As a preliminary remark, we report that the original informal specification document from which we began our formalization activity was only an early draft not even complete with respect to the ENEL method described in Section 2: it contained several ambiguities, inaccuracies, and inconsistencies. According to the double-spiral life-cycle model outlined in Section 1, the formalization process itself contributed to clarify many controversial issues so that a consistent informal document complying with ENEL standards was eventually obtained as a subset of the TRIO full specification document.

5.1 Structural versus Functional Modular Decomposition

Figure 12 depicts a view of the main components of the Advanced Supervisory System. It includes the above-mentioned diagnostic application PRECON, a component to manage the alarms (provided with its own log of alarms occurred in the past), a component to manage dynamic system

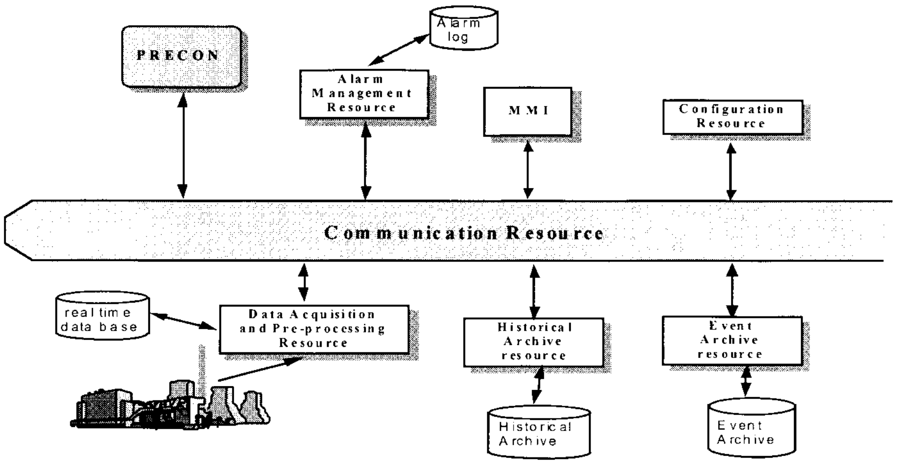

Fig. 12. Physical and hardware/software components of the supervisor.

reconfiguration, a Man-Machine Interface, a subsystem for acquisition of field data and measurements (stored in an *ad hoc* defined real-time data base), and databases of historical data and of the currently relevant events.

The diagram in Figure 13 reports the high-level modular structure of the TRIO specification.[7] Notice that all but one of the modules correspond to some physical system component. The only exception is the Global Plant Data Base (GPDB), which includes the specification of functions related to data storage and retrieval, but does not (necessarily) correspond to a system component. In fact, at the time when the specification was written it was not decided (and the specification intentionally left undetermined) whether all data of the supervisory system would be stored in a single centralized database or rather be dispersed in several repositories that could constitute a virtual, distributed database.

5.2 Global versus Local Statement of the Requirements

When a complex system is structured into a hierarchy of components and subcomponents, it is not always obvious which properties belong to the whole system and which ones are in the smaller scope of a single component. Thus, in the Supervisory System case study some properties regarding constraints on request, transmission, and delivery of data are stated in a global fashion with reference to the entire Supervisor System, whereas others refer more specifically to some of its components. For instance, a crucial requirement is that a query by PRECON to the GPDB must be answered within a given time bound. The axioms specifying this kind of properties belong to the outermost module *AdvancedSupervisorySystem*, because they do not regard a single component such as PRECON or

[7]For the sake of readability the connections among pairs of modules are reported as grouped into a single-arrow line: the actual specification included as many as 169 connections.

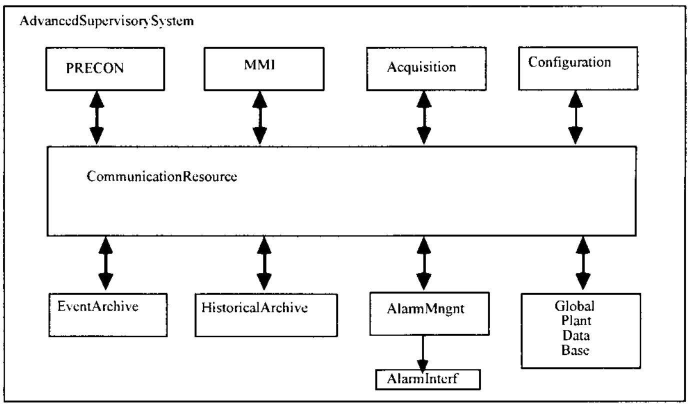

Fig. 13. High-level module decomposition of the supervisory system specification.

CommunicationResource, but several components must cooperate to reach the required result.

In the above example the following axiom modeling the time-out is placed in the *AdvancedSupervisorySystem* class. It asserts that the overall time of the operation is the sum of the times necessary for (a) delivering the message from PRECON to GPDB, (b) the internal processing of the GPDB, and (c) the data transfer in the opposite direction, and that it must not exceed 5 time units.

$$
\begin{aligned}
&\text{INqueryPR\_GPDB} \rightarrow \\
&\quad (\exists\, x\, \exists\, y\, \exists\, z\, (\, \text{Futr}(\text{OUTqueryPR\_GPDB} \wedge \\
&\qquad\quad \text{Futr}(\text{INresponse\_GPDB\_PR} \wedge \\
&\qquad\qquad \text{Futr}(\text{OUTresponse\_GPDB\_PR},\ z),\ y),\ x) \wedge \\
&\qquad x + y + z \leq 5))
\end{aligned}
$$

Notice that, due to the bottom-up approach which was adopted to maximize reuse of specifications, the sum of the maximum times specified for the components involved in the overall operation (i.e., PRECON, *CommunicationResource*, and GPDB) does not coincide with the total maximum time specified globally: they only must be *compatible*. Further analysis of the specification, devoted to establishing whether the requirements expressed in some internal modules can ensure the requirements stated globally, can be performed in successive phases of the development process, such as the specification validation, design, and verification (see Section 6).

5.3 Dealing with Fault Tolerance in the Communication Requirements

In Section 4.3 we illustrated means for expressing complex fault tolerance requirements on system performance, taking into account the number of occurred faults and fault repairs or the system robustness to disturbances on signals and values of variables. In the Advanced Supervisory System, the fault tolerance requirements were ubiquitous, though relatively simpler than in the previous examples. They took the typical form of relaxing some performance requirements (with respect to an idealized version that did not admit any fault) or strengthening them (if they initially considered the possibility that a fault would simply interrupt the service provided by the system). In this case, fault tolerance requirements were not factored out into special heir classes (there was no need to distinguish between instances with and without fault tolerance features); instead, several classes had their own section devoted to fault tolerance requirements.

Let us consider the *CommunicationResource* component: its typical requirement is to act as the carrier that takes a message from a component acting as a sender and delivers it to another component acting as a receiver, within a specified time bound. For instance a command c sent by PRECON (modeled by the event $InCommand(c)$) must be delivered to MMI (modeled by the event $OutCommand(c)$) within 5 time units. In this simplest, idealized form (no failure in the transmission is admitted) the property is expressed by the following axiom:

$$InCommand(c) \rightarrow WithinF(OutCommand(c), 5)$$

In practice one must take into account the possibility that, due to some fault, the communication resource is not currently available (modeled by the TD state *avail*); hence one must restate the requirement as in the following axiom

$$InCommand \wedge avail \rightarrow WithinF(OutCommand(c), 5)$$

in which the requirement has been relaxed by adding a conjunct to the premise of the implication. The requirement expressed by this last axiom can be strengthened by adding some fault tolerance property: in case the communication resource is not currently available the message needs not be delivered, but a notification of the failure must be sent within 10 time units (i.e., the time it takes the communication resource to recover from the fault) to both PRECON (modeled by the TD proposition *OutFailPR*), and MMI (modeled by the TD proposition *OutFailMMI*). This is stated by the following axiom

$$InCommand(c) \rightarrow \left(\begin{array}{c} WithinF(OutCommand(c), 5) \\ \vee \\ \left(\neg\, avail \wedge WithinF\left(\begin{array}{c} OutFailMMI \\ \wedge \\ OutFailPR \end{array} \right), 10 \right) \end{array} \right)$$

in which the proposition *avail*, taking into account the possibility of a fault in the transmission, has been moved to the conclusion of the implication (hence negated) and coupled with the fault tolerance requirement of failure notification.

5.4 Specifying Configuration Properties

The described system is required to be configurable to a large extent, both with respect to some parameters like the dimension of some physical components, or the value of some time constants (e.g., maximum delays), or the number and kind of some (inner module) components. All such forms of configuration have been specified using various forms of inheritance as illustrated in Section 4.4. Some configuration requirements are static (e.g., they define certain categories of plants and supervisory systems); others are dynamic (e.g., they define which elements belong to some sets; these sets may however be modified during operation).

Configuration information of this latter kind is typically coded using a predicate on the set of entities that can be configured: the predicate is true for a given element of this set (e.g., a variable is currently configured to be stored in the historical or event archives) if and only if the element belongs to the current configuration. Then, a given property can be restricted to the set of entities in the configuration by adding, in the axiom expressing it, a condition stating that the involved entities belong to the configured set.

For instance, the GPDB must send to the *HistoricalArchive* only the current value of the computed variables whose identifiers *IdCV* are in the current configuration of "interesting variables" (denoted by the predicate *ConfigIV*). This is expressed by a typical axiom schema reported below, where the property of interest is conditioned by means of an implication, whose premise refers to the current configuration:

$$\forall IdCV(ConfigIV(IdCV) \rightarrow sendResponseGPDB\_HA(current\_val(IdCV)))$$

6. PROPAGATION OF THE METHOD TOWARD THE OTHER LIFE-CYCLE PHASES

This article focuses on the first phases of the life-cycle, which are recognized to be the most critical ones. A brief look at other phases, however, can help the reader to appreciate the method we presented with reference to a broader context. We refer the reader to more specialized independent papers for a more thorough treatment of these topics.

6.1 Design and Implementation

It is a good tradition to clearly separate requirement specification from design, as stated by the classical "what versus how" motto. In practice the distinction is often not as sharp. In particular, the development of an SCS is strongly driven by the physical architecture of the system to be automated. Furthermore, the "double-spiral approach" emphasizes a view of the whole process as a sequence of refinement steps which evolves with some continuity from requirements to code implementation. In this article, we do not investigate the issue of separation between different phases of the life-cycle which may depend on collateral factors such as

—the separation of responsibilities and of know-how between people (usually the specifiers are application domain experts whereas implementors are computer people)

—the industrial tradition

—the adopted technology (the borderline shifted when moving from analogic technology to digital technology, and further to software from hardware control), and

—the adopted language.

However, let us notice that TRIO's class structure supports the refinement process through the typical "box explosion" mechanism which is adopted by most OO and function-oriented methods. Furthermore, its OO nature allows a natural and smooth pairing with other OO design methods devoted to lower-level phases of the life-cycle [Basso et al. 1998].

6.2 Validation and Verification

Any complex activity must be controlled to check that its outcomes actually accomplish its goals. This holds for the development of a whole application as well as for the subactivities into which it is usually partitioned. For instance, with reference to the traditional waterfall life-cycle model, we must check both that the requirement specification adequately captures real application needs (it is complete, consistent, etc.) and that the implementation, through several subphases, guarantees the stated requirements. The former type of control is referred to as requirement *validation* whereas the latter is called implementation *verification*.

Similarly to many other formalisms, TRIO can effectively support both validation and verification through suitable (semi)automatic tools. This is achieved by exploiting two major and fairly complementary techniques [Mok and Stuart 1996]:

—*Model generation* [Mandrioli et al. 1995]: Given a TRIO specification, sample models can be derived semiautomatically therefrom, providing examples of how a system complying with the given specification should behave. Conversely, a sample model can be confronted with a specification for compatibility. This technique can be applied both to validate the specification (specification prototyping) and to produce test cases to verify the correctness of the final implementation.

A more sophisticated application of this technique can even help system design at configuration time. In fact, recall from Section 4.4 that several heir classes of a given parent can often be defined by restricting some configuration parameter. Such subclasses can be subject to special requirements. In such cases we can use the model generation tool to check the *satisfiability* of additional requirements: in the positive case the tool provides configuration parameters that guarantee such requirements, therefore supplying a kind of automatic (configuration) design.

—*Theorem proving*: Given a set of TRIO *axioms* and a candidate TRIO *theorem*, a semiautomatic tool helps decide whether the candidate theorem can actually be derived from the axioms [Alborghetti et al. 1997]. This technique can be used to (a) validate specifications by deriving some logical consequences thereof to check whether they are consistent with user expectations; (b) verify that a lower-level specification fulfills the requirements stated in an upper level; and (c) verify that the code implementation is correct with respect to the requirement specification.

Consistently with the incrementality principle stated in Section 1, however, we consider such techniques not yet mature for a systematic adoption within industrial environments. Moreover, methods for their practical application are still lacking, though we already gained considerable and successful experiences in their use.

7. CONCLUDING REMARKS

We have now about 10 years of experience in the application of the TRIO formalism to the development of industrial projects. Most of these projects were developed within the long-term joint research carried on by ENEL-Ricerca and Politecnico di Milano, while a few involved also other industrial participants such as Italtel, Alcatel, and Sextant Avionique. The industrial projects on which TRIO was applied include the following:

—a new-generation digital energy meter,

—a dam static safety elaboration unit,

—the control system of a pondage power plant,

—the OpenDREAMS project reported in Section 5,

—the automation of a traffic light system, and

—a flight control system.

This shows, that besides the traditional fields of safety- or life-critical applications (nuclear, chemical plants, avionics, railway, automotive, weapon systems, etc.), an important field for application of formal methods is constituted by all areas where substantial economic investments are involved, because the cost of failures or of defects is very high: these include the fields of energy production and distribution and of flexible manufacturing systems. In some cases the cost of employing formal methods for the design of a device can be distributed over a largest number of installed units (e.g., the several millions of installed domestic energy meters); in other instances the price of even a short outage of some industrial plant cannot be tolerated; in some other cases the development of an industrial plant must be as much as possible independent from any technological platform, because of its very long lifetime (e.g., an energy distribution system can remain in operation for several decades, during which the technological platform on which its control apparatus is based may change entirely).

In all the above-mentioned projects the typical expected benefits of FMs were confirmed. Not only many more defects in specifications have been detected during the early phases than with traditional approaches, but the global project costs have been inferior to those with traditional methods even before going through new releases of the same product, when generality and reusability should come into play. In particular, Basso et al. [1998] report a comparative cost analysis between a TRIO-based and a traditional development of the control system of a pondage power plant which shows that the overall development cost of the TRIO-based approach is 15% less than that of the traditional approach. Moreover, Basso et al. [1998] show that the the budget repartition among the different phases is quite different in the two approaches: in the TRIO-based approach much effort is devoted in specyfing the system (twice as much) and in validating the requirements (five times as much), while the other phases showed a significant cost reduction. Thus, also, our experience is aligned with similar ones [Larsen et al. 1996] which confirms that the alleged increased costs of FMs are actually a false myth.

Thus, our experiences are in agreement with other reports on the application of FMs to real-life projects [Faulk and Heitmayer 1996; Harel et al. 1990; Lagnier et al. 1995].[8] We all admit, however [Faulk and Heitmayer 1996; Hinchey and Bowen 1995; Saiedian et al. 1996], that success depends heavily on the participation of, or strict cooperation with, the formalism experts, who are often the formalism developers themselves. Apart from a few noticeable exceptions we are still far from wide and autonomous acceptance of formal methods within an industrial environment.

In retrospective we find that several errors have been made by FMs advocates—including ourselves—when advertising their "products":

—Originally it was thought that good ideas by themselves were enough to convince practitioners to put them in practice. This misconception led to the "toy problem" objection. In this same realm it was thought that rough prototype tools were all that researchers had to deliver, overlooking the difference between using them in a small experiment and their application to real projects.

—Later, several researchers showed that real projects could benefit from FMs, but they did not pay much attention to technology transfer [Hinchey and Bowen 1995]: giving seminars and showing nice successful examples was not enough if attendants were not able to obtain similar results by themselves. According to the motto attributed to Rushby, FMs were actually often nice formalisms, possibly supported by promising tools, whose authors and experts (in most cases authors' pupils) were

[8]See also the Formal Methods Application Database, available at http://www.cs.tcd.ie/FME/.

able to manage quite profitably but which remained obscure for most outsiders. In other words they were lacking real methods.

In this article we have shown how we tried to solve this problem by augmenting the TRIO formalism with a specification method (or better, by enriching an informal specification method with a formal language), which should drive the user in the complex task of writing a requirement specification document. The focus of the article was not on the TRIO formalism, which is thoroughly presented and discussed elsewhere, but on the process of pairing a formalism with a method. We believe that most results of our experience could be generalized to other formalisms and specification methods though some mutual relationships are essential: for instance, major benefits have been obtained by a generalized OO attitude which permeates the whole process, from early informal system analysis down to implementation and verification. Furthermore, the original operational approach comprised in the ENEL specification standard required the introduction of a few corresponding constructs in the TRIO language. We also believe that most of our experience could be exploited even outside our original application field, i.e., SCS.

In agreement with the "V commandment" of Bowen and Hinchey [1995b], we maintain that the introduction of FMs, and of TRIO in particular, in the practice of industrial projects must be an *evolutionary rather than revolutionary* process. A major consequence of this principle is that our method recommends formalization after a first informal document has been written. This choice has no *a priori* motivation but derives from our wish to comply with existing standards and habits. In a possible further step we could conceive a method in which specifications are written directly in TRIO still keeping the structure described in Section 2.

Moreover, formality should be pursued incrementally also in the exploitation of its possible benefits. For instance, many advocates of FMs mainly advertise the level of confidence they can provide on system correctness thanks to the use of correctness proofs. However, developing the necessary skills to carry out formal correctness proofs—whether supported by sophisticated tools or purely manual—requires a much higher intellectual investment than learning to write a requirement through a mathematical formula rather than in English prose.

We apply an evolutionary approach even in our global technology transfer strategy. First, in most cases, applications are designed in cooperation with, or at least with the consultancy of, method developers, in agreement with the "IV commandment" of Bowen and Hinchey [1995b]. The cases of totally independent development are still a few exceptions and are limited to young people who are typically more fresh-minded and open to innovation. Second, we do not try to use all of our results at the first shot: initially we only wrote specifications in our language without trying any formal analysis at all; later, while we were trying to teach other people how to write specifications by themselves, we began producing test cases with our prototype tools. Presently, a small team of trained designers can manage our tools to analyze specifications through prototyping and to produce test cases; meanwhile, we developed (and are using experimentally) a more advanced tool aimed at proving properties both for specification and implementation analysis: we look forward to the time when we will be able to offer a complete methodology equipped with well-engineered tools which can be used autonomously by many industrial designers with reasonable training.[9] We also believe that real, deep innovation in design methodology can be obtained more easily in the long term, starting from the "bottom," i.e., through education at the college level and, as our own and other experiences seem to confirm, through the hiring of young, more adaptable and easy-to-train, engineers. For the time being, FMs could be more successful and gain more respect in the computing community by keeping formalisms experts in strict cooperation with application experts.

Finally, we share the view of many FM advocates that much promotion effort should be applied, and we hope to contribute to such an effort. We also suggest, however, that a *cooperative rather than competitive attitude* would be more productive: if our community devoted more effort to convince industrial people that FMs *per se* are useful in practice and less to argue on the relative pros and cons of different formalisms, FMs would perhaps gain better acceptance. For this reason the focus of this article has not been on advertising TRIO but on showing how a generic—suitable—formalism could be exploited within an industrial specification method.

ACKNOWLEDGMENTS

We want to thank the reviewers for their useful suggestions that helped us to significantly improve the quality of this article. We also want to thank all the people from the industry whose cooperation has been invaluable to the outcome of this research.

REFERENCES

ABRIAL, J. R. 1993. *Assigning Programs to Meanings*. Prentice-Hall, Englewood Cliffs, NJ.

ALBORGHETTI, A., GARGANTINI, A., AND MORZENTI, A. 1997. Providing automated support to deductive analysis of time critical systems. In *Proceedings of the 6th European Software Engineering Conference* (ESEC/FSE '97, Zurich, Switzerland, Sept. 22–25).

ALSPAUGH, T. A., FAUL, S. R., BRITTON, K. H., PARKER, R. A., PARNAS, D. L., AND SHORE, J. E. 1992. Software requirements for the A-7E aircraft. Tech. Rep. NRL9194. Information Technology Division, Naval Research Laboratory, Washington, DC.

[9]The use of sophisticated semantic tools is a typical example of the need for an accurate strategy that pairs technical innovation with organization and education plans: for instance, model checkers [Alur et al. 1990] can be used as "black boxes" with little or no knowledge of their internals; in most cases, however, semantic tools are interactive and require a fairly deep knowledge of the theoretical principles on which they are based. It was a nice surprise for the authors to find occasionally industries, such as Sextant Avionique, where a robust mathematical background was already available, which allowed them to use our prototype tools autonomously and effectively.

Alur, R., Coucoubetis, C., and Dill, D. L. 1990. Model checking for real-time systems. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*. IEEE Press, Piscataway, NJ, 414–425.

Anderson, T. and Lee, P. A. 1981. *Fault Tolerance—Principles and Practice*. Prentice-Hall, Englewood Cliffs, NJ.

Anderson, T. and Lee, P. A. 1990. *Fault-Tolerance—Principles and Practice, Dependable Computing and Fault-Tolerant Systems*. Springer Dependable Computing and Fault-Tolerant Systems Series, vol. 3. Springer-Verlag, New York, NY.

Basso, M., Ciapessoni, E., Crivelli, E., Mandrioli, D., Morzenti, A., Ratto, E., and San Pietro, P. 1998. A logic-based approach to the specification and design of the control system of a pondage power plant. In *Improving Software Practice: Case Experience*, Tully, C., Ed. John Wiley & Sons, Inc., New York, NY. Information about the project described in this paper can also be found—under the acronym ELSA—at http://www.esi.es/VASIE where results from selected ESSI projects are reported.

Benini, M., Ciapessoni, E., Fabiano, A., Morzenti, A., and Riccardi, L. 1996. PRECON application specification report. Rep. R22-V2, OpenDREAMS (Esprit no. 20843). Available by anonymous ftp at ftp-se.elet.polimi.it.

Bolognesi, T. and Brinksma, E. 1987. Introduction to the ISO specification language LOTOS. *Comput. Networks ISDN Syst. 14*, 25–59.

Bowen, J. and Hinchey, M. 1995a. Seven more myths of formal methods. *IEEE Softw. 12*, 4.

Bowen, J. and Hinchey, M. 1995b. Ten commandments of formal methods. *IEEE Comput. 28*, 4 (Apr.).

CEI-IEC. 1996. Industrial-process measurement and control: Evaluation of system properties for the purpose of system assessment. CEI-IEC International Standard 1069.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

Craigen, D., Gerhart, S., and Ralston, T. 1995. Formal methods technology transfer: Impediments and innovation. In *Application of Formal Methods*, Hinchey, M. and Bowen, J., Eds. Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Dürr, E. H. and Plat, N. 1995. VDM++ language reference manual. Inter. Rep. The Institute of Applied Computer Science.

Faulk, S. and Heitmayer, C., Eds. 1996. *Proceedings of the 11th Annual Conference on Computer Assurance*. (COMPASS '96, Gaithersburg, MD, June).

Faulk, S. R., Finneran, L., Kirby, J., Shah, S., and Sutton, J. 1994. Experience applying the CoRe method to the Lockeed C-130. In *Proceedings of the 9th Annual Conference on Computer Assurance* (COMPASS '94, Gaithersburg, MD, June).

Fitzgerlad, J., Larsen, P., Brookes, T., and Green, M. 1995. Developing a security-critical system using formal and conventional methods. In *Application of Formal Methods*, Hinchey, M. and Bowen, J., Eds. Prentice Hall International (UK) Ltd., Hertfordshire, UK.

Fuggetta, A., Ghezzi, C., Mandrioli, D., and Morzenti, A. 1993. Executable specifications with data-flow diagrams. *Softw. Pract. Exper. 23*, 6 (June), 629–653.

Gerhart, S., Craigen, D., and Ralston, T. 1994. Experience with formal methods in critical systems. *IEEE Softw. 11*, 1 (Jan.), 21–28.

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

Hall, J. A. 1990. Seven myths of formal methods. *IEEE Softw. 7*, 5 (Sept.).

Hall, A. 1996. Using formal methods to develop an ATC information system. *IEEE Softw. 13*, 2 (Mar.), 66–76.

Harel, D., Pnueli, A., Lachover, H., Naamad, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. 1990. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16*, 4 (Apr.), 403–414.

Heitmeyer, C. and Lynch, N. 1996. Formal verification of real-time systems using timed automata. In *Formal Methods for Real-Time Computing*, Heitmeyer, C. and Mandrioli, D., Eds. John Wiley & Sons, Inc., New York, NY.

Hinchey, M. and Bowen, J., Eds. 1995. *Application of Formal Methods*. Prentice Hall International (UK) Ltd., Hertfordshire, UK.

JONES, C. B. 1990. *Systematic Software Construction Using VDM*. Prentice-Hall, Englewood Cliffs, NJ.

LAGNIER, F., RAYMOND, P., AND DUBOIS, C. 1995. Formal verification of a critical system written in Saga/Lustre. In *Workshop on Formal Methods, Modelling and Simulation for System Engineering* (St. Quentin an Yvelines, France, Feb.).

LARSEN, P. G., FITZGERALD, J., AND BROOKS, T. 1996. Applying formal specification in industry. *IEEE Softw. 13*, 6 (May).

LARSEN, P. G., VAN KATWIJK, J., PLAT, N., PRONK, K., AND TOETENEL, H. 1991. SVDM: An integrated combination of SA and VDM. In *Proceedings of the Methods Integration Workshop*, P. Allen, A. Bryant, and L. Semmens, Eds. Springer-Verlag, New York, NY.

LUTZ, R. 1997. Reuse of a formal model for requirements validation. In *Proceedings of the 4th NASA Langley Formal Methods Workshop* (LFM '97, Hampton, VA, Sept.), C. M. Holloway and K. J. Hayhurst, Eds. NASA Langley Research Center, Hampton, VA.

MANDER, K. C. AND POLACK, F. A. C. 1995. Rigorous specification using structured systems analysis and Z. *Inf. Softw. Technol. 37*, 5 (May).

MANDRIOLI, D., MORASCA, S., AND MORZENTI, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst. 13*, 4 (Nov.), 365–398. A semantic analysis tool based on the techniques illustrated in this paper is available through anonymous ftp at ftp-se.elet.polimi.it.

MOK, A. AND STUART, D. 1996. Simulation vs. verification: Getting the best of both worlds. In *Proceedings of the 11th Annual Conference on Computer Assurance* (COMPASS '96, Gaithersburg, MD, June), S. Faulk and C. Heitmayer, Eds.

MORZENTI, A. AND SAN PIETRO, P. 1994. Object-oriented logical specification of time-critical systems. *ACM Trans. Softw. Eng. Methodol. 3*, 1 (Jan.), 56–98. An editing tool for the TRIO language is available through anonymous ftp at ftp-se.elet.polimi.it.

OMG. 1995. *CORBA: Architecture and Specification*. Object Management Group, Framingham, MA.

PARNAS, D. AND MADEY, J. 1995. Functional documentation for computer systems engineering. *Sci. Comput. Program. 25*, 41–61.

PETERSOHN, C., DE ROEVER, W. P., HUIZING, C., AND PELESKA, J. 1994. Formal semantics for Ward & Mellor's transformation schemas. In *Proceedings of the 6th Refinement Workshop of the BCS FACS*, D. Till, Ed.. Springer-Verlag, Vienna, Austria.

PFLEEGER, S. L. AND HATTON, L. 1997. Investigating the influence of formal methods. *IEEE Comput. 30*, 2 (Feb.), 33–43.

RAVN, A. P., RISCHEL, H., AND HANSEN, K. M. 1993. Specifying and verifying requirements of real-time systems. *IEEE Trans. Softw. Eng. 19*, 1 (Jan.), 41–55.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., LORENSEN, B., AND LORENSON, W. 1991. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ.

SAIEDIAN, H., BOWEN, J. P., BUTLER, R. W., DILL, D. L., GLASS, R. L., GRIES, D., HALL, A., HINCHEY, M. G., HOLLOWAY, C. M., JACKSON, D., JONES, C. B., LUTZ, M. J., PARNAS, D. L., RUSHBY, J., WING, J., AND ZAVE, P. 1996. An invitation to formal methods. *IEEE Comput. 29*, 4 (Apr.), 16–30.

SPIVEY, J. M. 1992. The Z Notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ.

STARTS. 1987. *The STARTS Guide, A Guide to Methods and Software Tools for the Construction of Large Real-Time Systems*. 2nd ed. National Computer Center Publications. Prepared by industry with the support of the DTI (Department of Trade and Industry) and NCC (National Computing Center).

WEBER, M. 1996. Combining statecharts and Z for the design of safety-critical control systems. In *Proceedings of Formal Methods Europe* (FME '96, Oxford, England, Mar.). Lecture Notes in Computer Science, vol. 1051. Springer-Verlag, New York, NY, 307–326.

Received: May 1997; revised: October 1997; accepted: March 1998