TRIO Specification of a Steam Boiler Controller

Angelo Gargantini and Angelo Morzenti
Politecnico di Milano, Dipartimento di elettronica e Informazione, Milano, ITALY
{garganti, morzenti}@elet.polimi.it

Abstract
We specify a controller for a steam boiler starting from an informal descriptions of its requirements. The specification is formalized in the temporal logic TRIO and its object-oriented extension TRIO+. To obtain a maximum of abstraction and reuse we make the specification parametric with respect to all equipment and hardware features, and we avoid to impose any particular strategy in the management of the available resources and in the control of the critical physical quantities.

1 Introduction

Computers are finding increasing applications in the fields of the control of real-time and safety-critical systems (avionic systems, medical systems, plant control systems, etc.). The development of such systems requires appropriate well-structured methods to master their high complexity. A particular importance is ascribed to the specification phase, since very often the errors encountered during their development can be traced back to inaccuracies or ambiguities in the description of the requirements. It is therefore particularly important that requirements specifications be precise (to avoid ambiguities), formal and mathematically well founded (to allow mechanized support in their analysis) and transparent (to serve as a common reference and a means of communication among humans).

The present report presents the specification of a steam boiler controller proposed in [AS] as a benchmark to assess the adequacy of specification methods to cope with practical non-trivial time- and safety-critical systems. The specification is written in TRIO, a temporal logic with metric on time that is particularly well suited to the specification of real time systems. TRIO is a logical language and therefore it favors a descriptive style where properties, rather than procedures or mechanisms, are specified, and the requirements are stated abstractly, avoiding any unnecessary bias with respect to particular design choices or implementation strategies. TRIO is the result of a long term cooperation among industry and academia, and in recent years a specification, validation, and verification environment has been built around the language, to support the development of industrially-sized time critical applications. The environment includes the definition of TRIO+, an object oriented extension of the language that effectively supports the modularization and the reuse of specifications of highly complex systems.

The report is organized as follows. Section 2 contains the formal specification of the steam boiler: it is organized in subsections according to the modular structure of the TRIO+ classes describing the overall system. Section 3 briefly illustrates how the specification can be usefully employed in the subsequent validation and verification activities with the support of automated tools developed around the

language, and discusses the notion of safety assessment of the specified system. Appendix 1 “(see CD-ROM Annex GM.1)” provides a brief overview of TRIO: to make the succeeding presentation reasonably self contained we illustrate the syntax of the language and the definition of the used derived operators; for the sake of brevity, the main features of the TRIO+ language are just recalled, referring the interest reader to the literature.

2. The Specification

To facilitate understanding by the reader, the presentation of the steam boiler formal specification in TRIO+ will follow a top-down approach. First, in Section 2.1, we illustrate the main assumptions and choices that we took in developing our specification. This should provide a rational to help the reader in obtaining a clear overall picture and a general understanding of our specification. Then in Section 2.2 we illustrate the modular structure of the specification describing informally how the various aspects of the requirements are separated and located in the specification components. At this point the reader should have precise and exact expectations on what will be found inside the modules at the lowest level of the part-of hierarchy determined by the modular structure, those containing the TRIO axioms that formalize the requirements. The detailed presentation of TRIO axioms is in Appendices from 2 to 11. “(see CD-ROM Annex GM.2)”

2.1 Assumptions and Choices

For the sake of abstraction the description of the steam boiler in the informal specification document [AS] deliberately leaves undetermined, and thus open to interpretation, several aspects of the control strategy and of the criteria to be used in the interpretation of messages coming from the equipment. Furthermore, each adopted specification formalism provides a particular notation to characterize the desired properties of the specified system, and different ways to obtain a model by abstracting away from irrelevant details. Most of the remarks listed below will be discussed in more depth in the subsequent paragraphs where the specification is presented in complete detail.

Representation of time. The informal specification document describes the operation of the program in terms of a possibly infinite iteration of a cycle that take place each five seconds. It is also assumed that: data transmission among the controller and the equipment is instantaneous and all messages are emitted (and received) simultaneously; that during every cycle the program can receive messages, analyze them, and send (response) messages. We model all these assumptions by choosing for our specification a temporal domain consisting of a discrete set, e.g., the set of integers, where each instant is intended to represent one distinct cycle time for the control program. As a consequence of this choice, the control program appears to have instantaneous reactions times, which is clearly a simplification of reality but is consistent with the abstraction level of the informal specification document [AS]. The main advantage of this choice is that the temporal properties and requirements can be described by means of very simple and transparent TRIO formulas. The description of the steam boiler at a more detailed level is obviously

possible by choosing a finer time granularity to represent time instants between consecutive program activations and inside each activation, but this would require to consider information regarding the Hw/Sw architecture of the implemented system and thus would involve the design phase, which we consider to be out of the scope of the present exercise.

Management of the pumps. The informal specification document [AS] does not describe any particular policy in the management of pumps (i.e., how to alternate the usage of the functioning pumps) and provides only a very simple criterion for the diagnosis of faults (i.e., how to establish that a pump and/or its controller is operating correctly). As a consequence we leave unspecified this choice when more pumps than necessary are available, by specifying only that, at any time, the controller must choose nondeterministically, among to functioning pumps, exactly those ones needed to cover the current requested throughput. Further refinement of the specification (or appropriate design choices) could specify a particular pump management policy (e.g., minimizing pumps wear by avoiding pump state changes, or balancing the load by alternating them as much as possible); in this case it would be necessary (and possible using the TRIO deductive system) to prove that such a policy is correct w.r.t. the high-level, nondeterministic specification of the present document.

Regarding the diagnosis of faults for pumps and pump controllers, we remark that even very complex and sophisticated diagnostic criteria cannot lead to absolute certainty on the effective state of the various equipment components if such criteria are based on comparisons among measures perceived through sensors and no assumption or estimation is made (as it happens in the document [AS]) on the availability and reliability of such measures coming from the sensors. In other terms, if *all* information coming from the field is equally subject to some uncertainty then all conclusions drawn from them based on comparisons or deductions (even though arbitrarily complex) cannot, in general, be absolutely secure. Keeping this remark in mind, for the diagnosis of pumps and pump controller faults we provide three sample criteria of increasing complexity, from the simplest one outlined in the [AS] document to two other, more sophisticated ones, where one considers the consistence between the state of each pump and that of its controller, or the probability of simultaneous faults. These more elaborate criteria can permit to improve the *average* effectiveness of the plant management but do not provide an absolutely error-free knowledge (and therefore control) of the plant state. These ideas will be illustrated and discussed in more depth in Appendix 7 “(see CD-ROM Annex GM.7)”.

Another consequence of the above remarks on the reliability of the information coming from the field is that a primary property such as *safety* (which in our case can be stated as: the controller will always go into the *emergency stop* mode as soon as the water level reaches the minimal or maximal limit quantity M_1 and M_2) will be asserted (and could be proved) only under the condition that the water level sensor only breaks down in a “recognizable” way, i.e., its indications are correct if they are reasonable, i.e., inside the physical limits for the capacity, 0 and C. In

section 3 we will also discuss the methodological implications of this approach in a correct and effective design discipline.

Operation during the rescue mode. For reasons related to the above remarks on the reliability of the information provided by the sensors, our specification prescribes, during the *rescue* mode, an operation of the plant that is more restrictive than that indicated by the informal specification document. When operating in the *rescue* mode, the controller abandons any information currently provided by sensors: it takes as reference the last useful value provided by the level sensor and evaluates the minimum time necessary for the plant to reach a dangerous condition considering it as out of control, i.e., it evaluates the minimum between the time needed to reach level M_1 when all pumps are closed with a maximal steam flow and the time needed to reach level M_2 when all pumps are open and no steam exits the boiler. If this time elapses without any event intervening that takes the controller out of the *rescue* mode, then it spontaneously goes into to *emergency stop* mode. The rationale for this stricter requirement on the controller behavior during the critical *rescue* mode is that reaching this mode is a symptom that something unexpected or unnoticed and potentially dangerous has happened, so the most prudent choice is not to rely on the sensors and actuators and work under the most pessimistic assumptions. Not surprisingly, this specification choice allows one to ensure the safety property under conditions that are particularly simple and easy to implement, as it will be shown in section 3. An alternative, less prudent plant operation during the *rescue* mode (such as the one described in the informal specification) would make the property of safety more difficult to obtain in practice, and also to prove formally as a property of the modeled system.

Management of the water level. Like the management policy for pumps, also the policy for keeping the water level within the prescribed limits is left undetermined in [AS]. An addendum to the informal specification simply suggests to open the pumps (without indicating the measure of such opening) if the water level is estimated to be below N_1, and to close them (again without indicating how much) when it is above the limit N_2. To obtain maximum generality we provide a framework where a variety of strategies for managing the water level can be described: at any time a quantity called “requiredThroughput” is defined as a non-negative real quantity whose actual value is determined by the adopted policy in water management. Then, in the present specification, we adopt a compromise between simplicity and effectiveness of the control policy, assuming that the control aims at keeping the water level as close as possible to the median level $(N_1+N_2)/2$, and consequently the pumps are opened (resp., closed) by a quantity which is essentially proportional (considering also the current estimated steam output) to the difference between the current and the desired water level. The definition of an optimum control algorithm for the water level falls in the area of control theory and is therefore considered out of the scope of the present exercise. As with many other features of the modeled system, the specification can be made parametric w.r.t. the policy for controlling the water level by means of the previously mentioned constructs of genericity and inheritance.

Errors in measuring. Every measurement is subject to some error, and the values obtained by the sensors for the water level and the steam flow can be no exception. The informal specification document [AS], however, does not mention possible inaccuracies in these measures. Consequently, we assume that the minimal and maximal limit quantities M_1 and M_2 for the water level are chosen in a conservative way as to account for any possible inaccuracy in the measurement of the controlled quantities, and thereafter we reason under the assumption that such measures are exact.

Modeling the environment. In the initial phase of requirements elicitation and formalization it is often very useful to model in the adopted formalism not only the device or system to be designed but also the environment where it will be put into operation when implemented. Therefore in our specification we model not only the controller but also the operator, the transmission system, the equipment, and the interactions among them. Then for the sake of brevity we mostly concentrate on the controller, since this component will be the actual object of the design activity. We point out, however, that the specification language can be usefully employed to describe relevant properties of the environment or of its interaction with the control program. In section 3 we provide an example thereof by describing a hypothesis on the functioning of the water level measuring device.

Parameters of the steam boiler plant. Several significant physical quantities of the equipment (such as boiler capacity, maximal and minimal limit and normal water quantity, maximal quantity and maximal gradient of increase or decrease in steam flow, nominal capacity of pumps) are mentioned in the informal specification document [AS], and the obvious fact that they may change from one plant to another is dealt with by indicating their value symbolically through suitable symbolic constants. Other parameters such as the number of pumps in the equipment, or the period of the operation cycle of the control program are probably assumed to be less likely to change and therefore are indicated as fixed values (there are 4 pumps and the cycle period is 5 seconds). To obtain maximum generality, flexibility and reusability of specifications (we could mention a “specifying for change” attitude) our specifications are generic and therefore parametric with respect to all the above mentioned quantities, which are subject to change due to the physical dimensioning of the various plants or are determined by design choices influenced by technological factors (e.g., the cycle period).

Description of the initialization mode. We found it convenient to distinguish three phases in the initialization mode (i.e., *waiting*, *adjustingWaterLevel*, *programReady*) to describe in a simpler and more explicit way the sequence of actions carried out by the control program when operating in this mode. To fully comply with the informal specification document [AS], however, this separation of phases is confined in an inner module of the controller component and does not emerge in the communication between the controller and the equipment.

2.2 Modular Structure of the Specification

Object oriented methodology comprises classical modularization criteria, therefore our specification is divided into modules according to the principles of

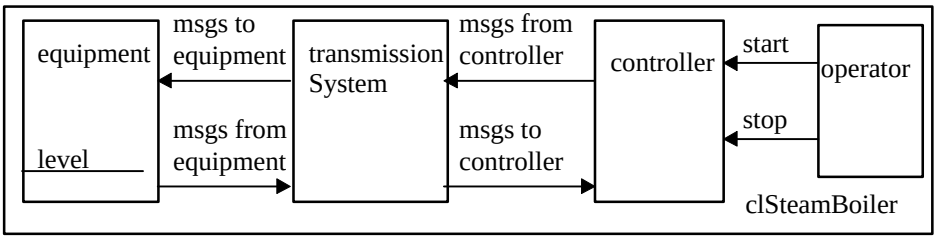


Fig. 1 Graphical representation of class clSteamBoiler

encapsulation and information hiding, separation of concerns, maximization of intra-module cohesion and minimization of inter-module coupling (and hence of module interfaces). Moreover, the typical object-centered view (emphasizing the physical and logical components of the system) is blended with a more functional view (modules called *controller*, *management*, *diagnosis* are introduced) so that system functions can be described in a general and abstract way. As it often happens, the modular structure supports abstraction and reuse, but it also facilitates presentation and understanding, therefore the specification is highly structured, especially in the part regarding the pumps.

The highest level in the module hierarchy is shown in Fig. 1, representing class *steamBoiler* that includes modules for the system physical components (*equipment*, *transmissionSystem*, *controller*, and *operator*) and the connections among them consisting of the information exchanged and the delivered commands. In Appendix 2 "(see CD-ROM Annex GM.2)" we report the detailed graphic representation and the textual declaration of the same class. It can be noticed that in the *equipment* module the local item *level* models the actual physical water level, which is in principle distinct from the measured level as perceived by the controller through the sensors. Modeling as separate entities the actual and the measured water level will allow us, in Section 4, to formalize some remarks on the reliability of performed measures and on the safety of the control algorithms based on them.

For brevity we do not model other features of the environment, and in the remaining parts of the present specification we focus on the controller, whose structure is shown in the graphical representation of class *clController* of Fig. 2, and whose textual declaration is in A.3 "(see CD-ROM Annex GM.3)". This figure shows the components *steam* and *level*, which concern the measurement and control

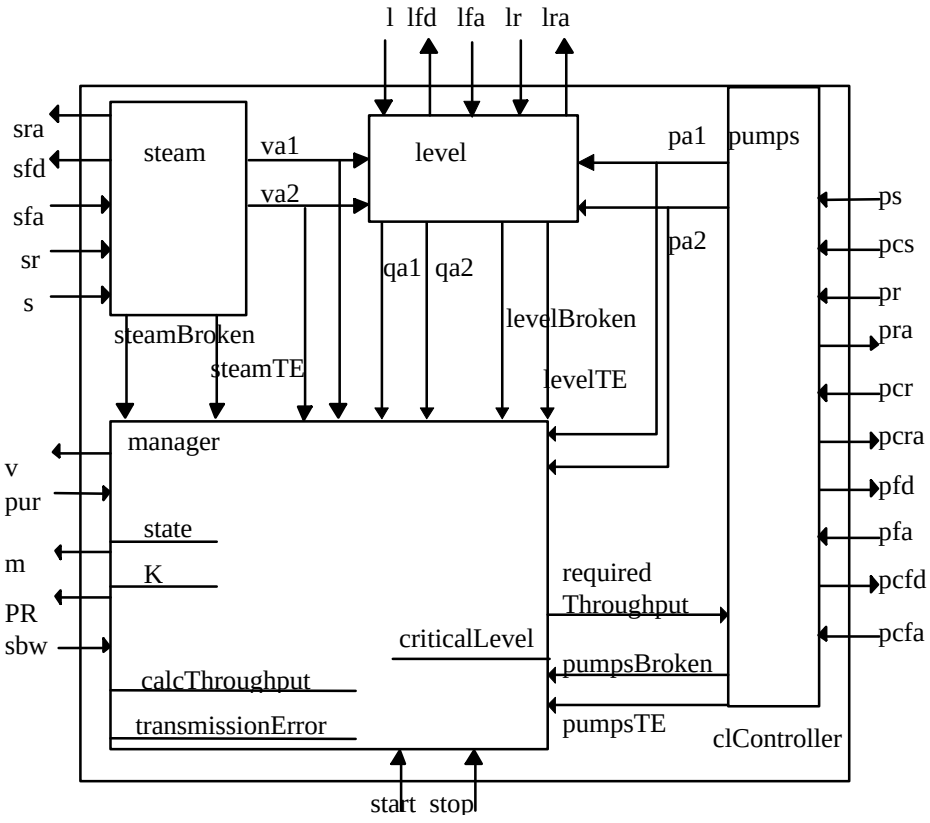


Fig. 2 The class clController

of the quantity of water in the boiler and of the exiting steam. Module *pumps* specifies control and management of the pumps. Module *manager* specifies the operation of the control program, as described in Section 3 of the informal specification document [AS], with all actions to keep the water level within the required bounds and to face sensor and actuator faults by operating in the *degraded* or *rescue* mode.

Fig. 3 depicts the clPumps: its textual version is in Appendix 5 "(see CD-ROM Annex GM.5)": it includes an array of modules, called *pumpSet*, containing instances of class *clPumps* in a number equal to the number of pumps actually present in the plant (notice that the specification is generic with respect to this number). The *pumpManager* component includes the specification of how to govern pumps, i.e., the indication of pumps opening or closing depending on the current required throughput and on the estimated current state of the pumps. Let us anticipate that the commands to the pumps, the diagnosis of their state, and the messages sent to the equipment are determined in the *clPump* class, which will be described briefly in the next section.

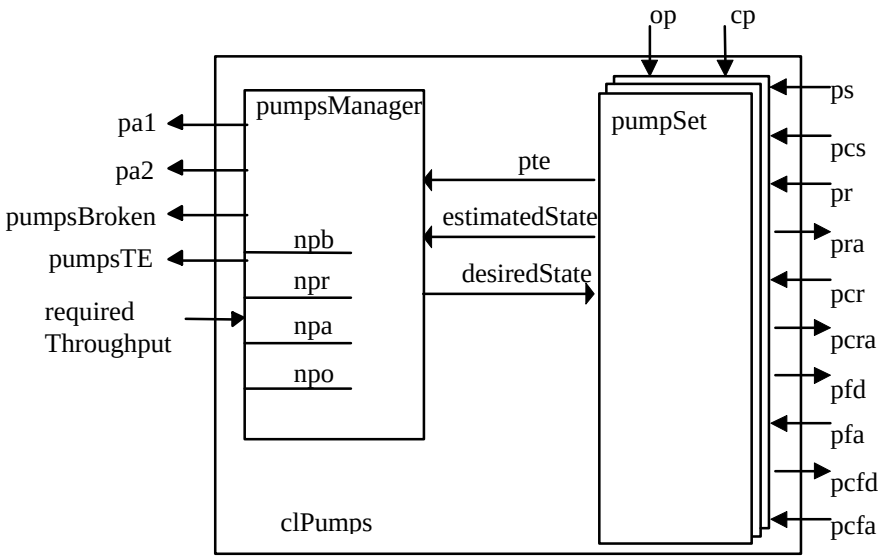


Fig. 3 The class clPumps

2.3 The *Pump* Module

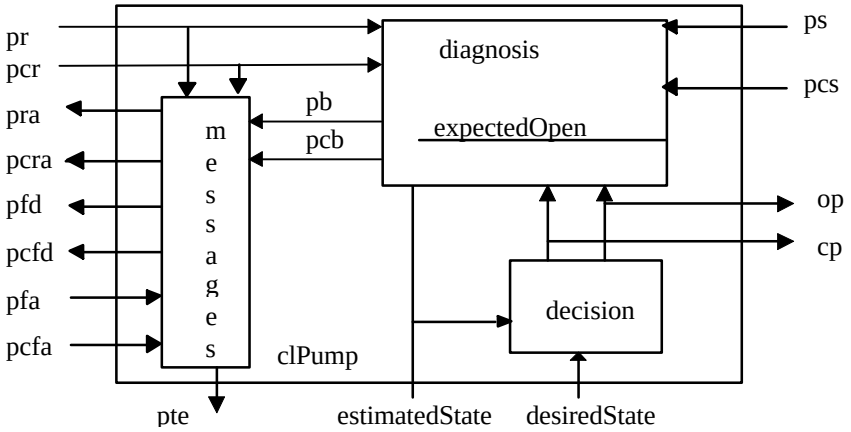


Fig. 4 The class clPump

Fig. 4 reports the graphical representation of the class *clPump*. The textual declaration can be found in Appendix 6 "(see CD-ROM Annex GM.6)". This class models a single pump of the plant, and there are many instances of it in the array of modules included in class clPumps (4 in the case considered by [AS]). The class is organized into three modules: module *diagnosis* specifies how faults of the pump or of its controller are determined; module *decision* characterizes the opening or closing commands to the pump according to its estimated state and to its desired state as determined by the pumpManager module; module *messages* defines the messages to be exchanged with the equipment regarding faults and repairs, in interaction with the diagnosis module.

Performing diagnosis on the state of the pumps and its controller is a crucial operation because the correct plant operation and control depends on the accuracy with which the actual state of the various devices can be estimated. For this reason we specify three possible ways of performing this operation. Here we report only the first one, that is simply a formalization of the criterion reported in [AS].

pumpDiagnosis:

$$pb \leftrightarrow \left(\begin{array}{c} UpToNow(pb) \wedge \neg pr \vee \\ expectedOpen \wedge ps(closed) \vee \\ \neg expectedOpen \wedge ps(open) \vee \\ \left(\begin{array}{c} UpToNow(\neg pcb) \vee \\ pcr \end{array} \right) \wedge \left(\begin{array}{c} ps(closed) \wedge pcs(open) \vee \\ ps(open) \wedge pcs(open) \end{array} \right) \end{array} \right)$$

pumpControlDiagnosis:

$$pcb \leftrightarrow \left(\begin{array}{c} UpToNow(pcb) \wedge \neg pcr \vee \\ expectedOpen \wedge pcs(closed) \vee \\ \neg expectedOpen \wedge pcs(open) \vee \\ \left(\begin{array}{c} UpToNow(\neg pb) \vee \\ pr \end{array} \right) \wedge \left(\begin{array}{c} ps(closed) \wedge pcs(open) \vee \\ ps(open) \wedge pcs(open) \end{array} \right) \end{array} \right)$$

We propose other two ways for pump fault diagnosis in Appendix 7 "(see CD-ROM Annex GM.7)". The complete specifications of *decision* and *messages* modules are respectively in Appendix 8 "(see CD-ROM Annex GM.8)" and in Appendix 9 "(see CD-ROM Annex GM.9)".

2.4 The *Level* and *Steam* Modules

The classes *clSteam* and *clLevel*, which we report in Appendix 4 "(see CD-ROM Annex GM.4)", specify operations similar to those described for the pumps by the class *clPumps*, i.e., operations regarding monitoring of the device state, exchange of messages with the equipment regarding faults, detection of transmission faults, and computation of estimated values for the water level and the exiting steam.

2.5 The *manager* Module

The class *clManager*, reported Appendix 11 "(see CD-ROM Annex GM.11)", specifies the module *manager* of Fig. 2 and describes general operations regarding the various modes of operation, the detection of transmission errors, and the government of the water level.

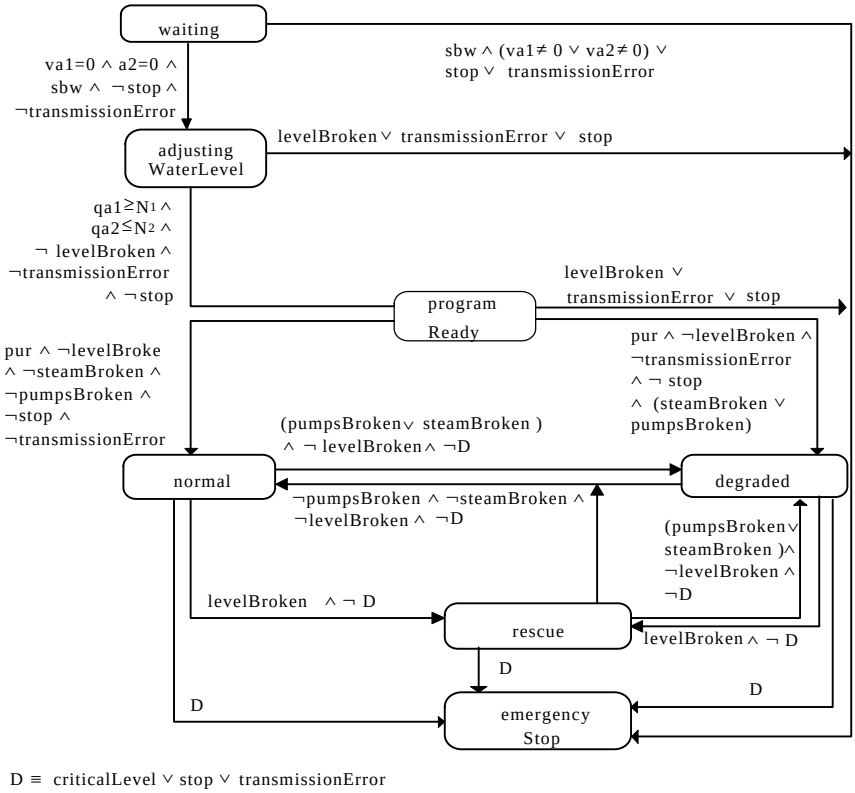


Fig. 5 Finite state automaton for the transitions among modes of the controller

We adhere to the description of the *modes* of operation employed in the informal specification document, therefore we model the principal structure of the

control program as a finite state machine, whose states and transitions are represented in Fig. 5. The transitions of this automaton can be formally described in TRIO in a rather obvious and uniform way, which we just exemplify with this axiom:

normalToRescue

$$\left(\begin{array}{c}\text{UpToNow}(\text{mode}(\text{normal})) \wedge \\ \text{levelBroken} \wedge \\ \neg\, \text{D}\end{array}\right) \rightarrow \left(\begin{array}{c}\text{mode}(\text{rescue}) \wedge \\ \text{Until}\big(\text{mode}(\text{rescue}), \neg \text{levelBroken} \vee \text{D}\big)\end{array}\right)$$

3. Use of the Specification

The obvious purpose of any specification is to express requirements and to serve as a reference for the successive phases of design, implementation, verification, and maintenance. Before these are undertaken, a very useful activity is often performed (especially when the specified systems are particularly complex or critical), namely the validation activity, which consists of establishing whether the actual requirements were indeed captured and correctly expresses by the specification. Formal methods, being based on a solid mathematical foundation, have a clear and unambiguous semantics, so that the validation and verification activities can be effectively supported by (semi)automatic software tools that can greatly enhance the effectiveness and the practical impact of such activities. This is the case with TRIO, where an environment of tools for editing specification, validating them and verifying design and implementation has been developed in recent years at Politecnico di Milano.

Broadly speaking, the validation activity can take in TRIO the form of *history checking*, *history generation* (i.e., *simulation*), and *property proving*.

When performing history checking [F&M94] the designer invents (with the aid of a suitable tool) histories of the modeled system (i.e., sequences of events, system configurations, and values for the significant quantities that represent a hypothetical trace of a system execution) that in his/her view correspond to a possible behavior of the specified system where the requirements and properties are apparent. For instance, possible histories of the steam boiler could include sequences of faults in the pumps, and the actions of the controller to deal with them, possibly in presence of particularly high (or reduced) steam production. Such histories are then checked, i.e., confronted for consistency with the specification, by considering each history as the frame of an interpretation structure for the TRIO formulas. The results of history checking are useful both to the final user, who verifies that his/her expectations on the system behavior are sensible, and to the specifier, who controls that his/his understanding of the requirements are correct and have been effectively formalized by means of the formal notation.

A more sophisticated method of validation consists of simulating the modeled system by generating (with the support of suitable specialized interpreters, see [MMM95]) histories of the specified systems under particular constraints that may represent an initial system configuration or particular combination of input events coming from the environment and are assumed to stress particular system functionalities that the designer wants to explicitly visualize.

The most complex, general, and effective validation activity is obtained by proving properties that are supposedly ensured by the requirements as expressed in the formal specification. From a logical viewpoint, as it happens in the case of TRIO, such properties are theorems that are derived in a theory consisting of the TRIO general axioms augmented with the axioms that are included the specification document. The derivation of such theorems can be made manually using the axiomatic system presented in [FMM94] or with the support of a theorem proover, such as PVS, where the TRIO semantics and axiomatic system have been suitably encoded, as it was done in [Jef95]. Typical properties that one would like to derive for a time critical system would be liveness, absence of deadlock, or the ability of the system to control the environment by maintaining invariant in time a given configuration or relation among components or physical quantities. As an example thereof, we would like to express in TRIO a property of (physical) safety that was never explicitly formalized in the specification presented in the preceding sections, but is clearly implied as the main purpose of the designed controller. As anticipated in the remarks of section 2.1, we state such safety requirement under suitable assumptions regarding the correct functioning of the measuring and transmission devices that the controller uses during operation. A first assumption is that the transmission system component is correct (although not necessarily permanently available), that is, any received data are equal to those transmitted. This assumption is easily formalized through the following simple TRIO formula

$$\mathrm{l}(v) \rightarrow \mathrm{lE}(v)$$

asserting that if a value is received by the controller then it is the same value that has been measured (and then sent) by the physical equipment.

The second assumed property regards the water level measuring device and asserts that when it is broken (i.e., its measurement is significantly different from the actual water level) then it gives a value that is out of the possible range of measures. This is formalized is TRIO as follows

$$\mathrm{lE}(v) \wedge \mathrm{v\_equipment.level} \rightarrow (v<0 \vee v>C)$$

or equivalently, and perhaps in a more intuitive manner, as

$$\mathrm{lE}(v) \wedge 0 \leq v \leq C \rightarrow v = \mathrm{equipment.level}$$

This assumption ensures that the measurements from the water level sensor are reliable, in that if they are incorrect then they are out of range and thus can be immediately recognized as such. Under the two above hypotheses the fundamental property of physical safety can be stated as follows.

$$\forall v \left(\begin{array}{c} (\mathrm{l}(v) \rightarrow \mathrm{lE}(v)) \\ \wedge \\ (\mathrm{lE}(v) \wedge 0 \leq v \leq C \rightarrow v = \mathrm{equipment.level}) \end{array} \right) \rightarrow \left(\begin{array}{c} \neg(M_1 < \mathrm{equipment.level} < M_2) \\ \rightarrow \mathrm{m}(\mathrm{emergencyStop}) \end{array} \right) \quad (\ddagger)$$

The above formulas implies that the controller will go into the emergency stop mode before the water level reaches the extreme levels M_1 and M_2 that could be dangerous for the plant.

For the sake of brevity we will not go into the formal proof of the above property, but we report some remarks on the degree of confidence possibly provided by such proofs (such remarks are intended as an illustration and complement of those reported in section 2.1). The hypotheses under which the safety property holds (as expressed by the premise of the external implication in formula ‡) are not themselves absolutely true, because sensors are, like any other physical device, subject to wear and could become unreliable, while communication channels are subject to (at least) transient transmission errors due, e.g., to interferences. There exist however techniques for obtaining fault tolerance in the measure of physical quantities (e.g., device redundancy and majority voting) and in data transmission (e.g., theory of self correcting codes and protocols) such that the *probability* of faults can be made arbitrarily low (of course at the price of devoting sufficient resources to that purpose). As a result, the controlled plant can be shown to be safe "with a probability 1-ε": this does not mean that the control algorithm or the proof of its correctness could be wrong, but that there exists a certain degree of uncertainty on the used input data.

In the TRIO framework, verification can take the two main forms of formal proofs and testing. Performing the formal proof of the correctness of an implementation w.r.t. its implementation requires that all relevant features of the hardware system software employed are themselves formalized, as outlined in [M&M94], and such proofs can be performed, as noted above, both by hands or with the support of a theorem prover.

Several years of experience of the authors in cooperation between academia and industry have however shown that formal proofs are not considered as a practically convenient technology, whereas various forms of testing are much more appreciated and widely employed. Formal specifications in TRIO can be used to support the generation of functional test cases, that include both data to be input as stimuli to the system to be tested, and expected results of the experiments, to be compared with the actual reactions of the tested system. Functional testing (often referred to in handbooks as *black box* testing) verifies the requirements of the system under test without any reference to the actual hardware/software structure of the implementation; it should be considered a complement, not an alternative, to structural, white box testing, which is based on the structural properties of the implementation (e.g., control flow in software code). The activity of test case generation from TRIO specification is supported by a tool based on interpretation algorithms similar to those for system simulation and, moreover, support the annotation of test cases with information useful for the testing experiment [MMM95]. The algorithms for test case generation are computationally intensive, so that their practical employment can become unfeasible for large formulas, like those occurring in specifications of industrially sized applications. For this reason, strategies for generating test cases starting from TRIO+ specification have been defined, that exploit the modular structure of specifications and take advantage of

the direction of the connections among modules to follow logical or functional dependencies among data, thus making the generation process more easy, transparent to the user, and effective [MMS96].

The specification method through TRIO+ classes exemplified in section 3 and the validation and verification activities discussed in the present section have been applied to several case studies [CSt90, CSt92] and industrial projects: among these we mention a project concerning the monitoring and control through semaphore systems of surface traffic in densely populated urban areas [NUS05] and an application (whose development was carried out in the framework of an ESSI project) regarding the balancing of the load among power generators in a pondage power plant of ENEL, the Italian energy board [BC&95].

The latter application of the TRIO language and method covered all system development phases, from requirement elicitation down to design and coding. In particular, it is interesting to consider how system design was obtained by systematic transformation and refinement of the specification. The software implementation was obtained by coding the features described in the main TRIO+ class and its composing modules into a set of modules of MOOD [Bas94, MR94], an object oriented language for real-time concurrent system design. During the design particular attention was devoted to maintaining a close correspondence between the specification objects and the design objects. This was highly facilitated by the fact that both TRIO and MOOD are based on object-oriented paradigms. The refinement of specification into executable code was obtained systematically by transforming the TRIO+ classes into MOOD classes, whose local variables were directly derived from the local items of the corresponding TRIO+ class; moreover, TRIO+ connections were implemented as asynchronous communication channels among MOOD classes (a construct directly available in the chosen programming language). It is to be noticed that the fastest required reaction times of the implemented systems were of the order of a few milliseconds, while in the steam boiler system the controller activation cycle is assumed to be 5 seconds, therefore the above described, trivial implementation technique could be certainly applied to the present case.

4. Evaluation and Comparison

Here we answer a few questions asked by the editors of the book with the purpose of providing some comparison criteria among the various solutions to the proposed problem.

1. The controller component is specified in full detail, while the other two components (the physical plant and the transmission system) are left unspecified. The controller is specified both in a formal and rigorous way, by means of TRIO formulas, and verbally, by means of suitable comments in natural language attached to the formulas. Note that the specification is parametric with respect to several features of the controller, such as the number of pumps, the physical constants of the plant, and the strategies for fault diagnosis in the pumps.

2. No; TRIO is a formal notation mainly devoted to requirements elicitation and specification, system validation and verification, therefore no specific guidelines are provided, in general, to support design and implementation.
3. Among the numerous proposed solutions those who are more closely comparable to ours are those that share its overall approach, i.e., use a descriptive notation (e.g., mathematical logic) to specify the requirements in the form of constraints or properties but do not provide any indication of the system architecture nor its implementation of terms of state-transition systems. Among these we mention [S], "see Chapter S, this book" which uses an equational style of specification and does not provide specific indications for the design and implementation, [OKW], "see Chapter OKW, this book" which extensively adopts object-oriented concepts and methods, [CW1] "see Chapter CW1, this book" [LM], "see Chapter LM, this book" and [LW], "see Chapter LW, this book" which are based on (temporal) logic, [BCPR] "see Chapter BCPR, this book" and [GDK], "see Chapter GDK, this book" which adopt a specification style based on an implicit state and use extensively first order logic to describe system properties.
Conversely, the complementary solutions are those where an operational approach is adopted, whereby the system is modeled by means of a (possibly abstract) state machine and its behavior is described through next state functions that in some case are characterized in a rather elaborate or indirect way. Among those we mention [BBGDR] "see Chapter BBGDR, this book" and [HW], "see Chapter HW, this book" which are based on a state transition machine providing immediate support to design and implementation, [VH] "see Chapter VH, this book", which adopts an operational approach with refinements and an automatic support to proof of consistency among different refinement levels, [CD] "see Chapter CD, this book" and [DC], "see Chapter DC, this book" which are based on an execution model rather closed to a programming language, [WS], "see Chapter WS, this book" which, being based on process algebras, is directly executable.
4. We spent approximately 1 person week to read and understand the informal description document [AS], and 2 person weeks to write the specification in TRIO+. We believe that a general answer to this question can be hardly provided. The time and effort needed to learn the TRIO language and method depends significantly on the educational background of the learner. A few hours could suffice a person with a deep knowledge of first order mathematical logic, while there could be no upper limit for a refractory programmer who has a very limited mathematical background.
5. A rather limited knowledge of TRIO and TRIO$^+$ should suffice, since only the basic operators and constructs are applied in the proposed solution.
A precise answer to the questions b. and c. meets the same difficulties as in point 4. above, and similar remarks apply.

References

[Bas94] M. Basso, MML Object Oriented Design Metodology Reference, TXT- Ingegneria Informatica, 1994

[BC&95] M.Basso, E.Ciapessoni, E.Crivelli, D.Mandrioli, A.Morzenti, E.Ratto, P.San Pietro, “Experimenting a Logic-Based Approach to the Specification and Design of the Control System of a Pondage Power Plant”, ICSE-17 Workshop on Industrial Application of Formal Methods, Seattle, WA, April 1995.

[CSt 90] Specification environments for real time systems based on a logic language, Technical annex to research contract 27/90, December 1990, Case studies (in Italian) on a regulator in a pondage power plant and on high-voltage substations, .

[CSt 92] *Specification environments for real time systems based on a logic language*, Technical annex to research contract 49/92, December 1992, Case studies (in Italian) on a programmable digital energy and power meters and on data collection and elaboration for dam security, .

[F&M94] M.Felder, A.Morzenti, “Validating real-time systems by history-checking TRIO specifications”, ACM TOSEM-Transactions On Software Engineering and Methodologies, vol.3, n.4, October 1994.

[FMM94] M.Felder, D.Mandrioli, A.Morzenti, “Proving properties of real-time systems through logical specifications and Petri net models”, IEEE TSE-Transactions of Software Engineering, vol.20, no.2, Feb.1994, pp.127-141.

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

[Jef95] R.D.Jeffords, “An Approach to Encoding the TRIO Logic in PVS”, Technical Report, Naval Research Laboratory, Wash.,D.C.,1995.

[M&M94] L.Mezzalira. A.Morzenti. “Relating specified time tolerances to implementation performances”, 6th IEEE Euromicro Workshop on real-time systems, Vaesteraas, Sweden, June 1994.

[M&S94] A. Morzenti, P. San Pietro, “Object-Oriented Logic Specifications of Time Critical Systems”, ACM TOSEM - Transactions on Software Engineering and Methodologies, vol.3, n.1, January 1994, pp. 56-98.

[MMM95] D.Mandrioli, S.Morasca, A.Morzenti, “Generating Test Cases for Real-Time Systems from Logic Specifications”, ACM TOCS-Transactions On Computer Systems, November 1995.

[MMS96] S.Morasca, A.Morzenti, P.San Pietro, “Generating Functional Test Cases in-the-large for Time-critical Systems from Logic-based Specifications”, Proc. of ISSTA 1996, ACM-SIGSOFT International Symposium on Software Testing and Analysis, January 1996, San Diego, CA, U.S.A.

[MR94] Architetture e componenti software riusabili ad alta tolleranza ai guasti, Research Report, MilanoRicerche 1994.

[NUS95] NUS (Sistemi per la Pianificazione Urbana e territoriale), “Detailed specification of a traffic monitor and a semaphore regulator”, Project Documentation (in Italian), October 1995.

Appendices

A.1. A Brief Overview of TRIO and TRIO+

TRIO is a first order logical language, augmented with temporal operators which permit to talk about the truth and falsity of propositions at time instants different from the current one, which is left implicit in the formula. Unlike classical temporal logic, TRIO allows the specifier to express strict timing requirements by means of two basic operators—*Futr* and *Past*—which refer to time instants whose distance, in the future or in the past, is specified precisely and quantitatively. We now briefly sketch the syntax of TRIO and give an informal and intuitive account of its semantics; detailed and formal definitions can be found in [GMM 90].

The alphabet of TRIO is composed of variable, function, and predicate names, plus the usual primitive propositional connectors '$\neg$' and '$\rightarrow$', the derived ones '$\wedge$', '$\vee$', '$\leftrightarrow$', ..., and the quantifiers '$\exists$' and '$\forall$', and plus temporal operator symbols *Futr* and *Past*. The language is typed, in that a domain of legal values is associated with each variable, a domain/range pair is associated with every function, and a domain is associated with every argument of every predicate. Among variable domains there is a distinguished one, called the *Temporal Domain*, which is numerical in nature: it can be the set of integer, rational, or real numbers.

Variables, functions, and predicates are divided into *time dependent* and *time independent* ones. This allows for representing change in time. Time dependent variables represent physical quantities or configurations that are subject to change in time, and time independent ones represent values unrelated with time. Time dependent functions and predicates denote relations, properties, or events that may or may not hold at a given time instant, while time independent functions and predicates represent facts and properties which can be assumed not to change with time.

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

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

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

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

are TRIO formulas too, that are satisfied at the current time if and only if property *A* holds at the instant which is *t* time units ahead (resp., behind) the current time. On the basis of the primitive temporal operators *Futr* and *Past*, numerous derived operators can be defined for formulas, including the following list.

Operator	*Definition*	*Explanation*
AlwF(A)	$\forall t\ (t > 0 \rightarrow Futr(A, t))$	A will always hold
AlwP(A)	$\forall t\ (t > 0 \rightarrow Past(A, t))$	A has always held
Alw(A)	$AlwP(A) \wedge A \wedge AlwF(A)$	A always holds
Som(A)	$\neg$ Alw ($\neg$ A)	Sometimes A holds
Lasts(A, d)	$\forall d'(0<d'<d \rightarrow Futr(A, d'))$	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
Until(A_1, A_2)	$\exists t\ (t>0 \wedge Futr(A_2, t) \wedge Lasts(A_1, t))$	A_1 will hold until A_2 starts to hold
Since (A_1, A_2)	$\exists t\ (t > 0 \wedge Past\ (A_2, t) \wedge Lasted\ (A_1, t))$	A_1 held since A_2 became true
UpToNow (A)	$\exists \delta\ (\delta > 0 \wedge Past\ (A, \delta) \wedge Lasted\ (A, \delta))$	A held for a nonnull time interval that ended at the current instant
Becomes (A)	$A \wedge UpToNow\ (\neg A)$	A holds at the current instant but it did not hold up to now

Notice that for the operators expressing a duration over a time interval (for example Lasts) we gave definitions where the extremes of the specified time interval are excluded, i.e. the interval is open. Operators including either one or both of the extremes can be easily derived from the basic ones we listed above. For notational convenience, in order to indicate inclusion or exclusion of the lower or upper bound of the interval, we append to the operator's name two subscripts, 'i' or 'e', respectively. For example, $Lasts_{ie}$ and $Since_{ie}$ are defined as follows.

$Lasts_{ie}\ (A, t)$ $\quad Lasts\ (A, t) \wedge A$

$Since_{ie}\ (A_1, A_2)$ $\quad \exists t\ (t > 0 \wedge Past\ (A_2, t) \wedge Lasted\ (A_1, t) \wedge Past\ (A_1, t))$

TRIO has proved to be an adequate language for specifying real-time systems features. However, its use becomes difficult when considering large and complex systems, because TRIO specifications are very finely structured: the language does not provide powerful abstraction mechanisms, and lacks an intuitive and expressive graphic notation.

To support specification in the large, we enriched TRIO with concepts and constructs from object oriented methodology, yielding a language called $TRIO^+$ [M&S94]. Among the most important features of $TRIO^+$ are the ability to partition the universe of objects into classes, inheritance relations among classes, and mecha-

nisms such as genericity to support reuse of specification modules and their top-down, incremental development. TRIO$^+$ maintains an intensional notion of object identity: we require an object described by a TRIO$^+$ specification to be identified with the history of its evolution, and an instance of a class for a composite system to include instances of the system's components. Structuring the specification into modules supports an incremental, top-down approach to the specification activity through successive refinements, but also allows one to build independent and reusable subsystem specifications, that could be composed in a systematic way in different contexts. Also desirable is the possibility of describing the specified system at different levels of abstraction, and of focusing with greater attention and detail on some more relevant aspects, leaving unspecified, or less formalized, other parts that are considered less important or are already well understood.

TRIO$^+$ is also endowed with an expressive graphic representation of classes in terms of boxes, arrows, and connections to depict class instances and their components, information exchanges and logical equivalence among (parts of) objects. In principle, the use of a graphic notation for the representation of formal specifications does not improve the expressiveness of the language, since it provides just an alternative syntax for some language constructs. In practice, however, the ability to visualize constructs of the language and use their graphic representation to construct, update or browse specifications can make a great difference in the productivity of the specification process and in the final quality of the resulting product, especially when the graphic view is consistently supported by means of suitable tools, such as structure-directed editors, consistency checkers, and report generators.

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

A TRIO$^+$ specification is built by defining suitable *classes*. A class is a set of axioms, describing the system, constructed in a modular, independent way, following information hiding principles and object oriented techniques. Classes may be *simple* or *structured*, may be *generic* and may be organized in *inheritance* hierarchies.

A *simple class* is a group of TRIO axioms, preceded by the declaration of all occurring predicates, variables, and functions. A class may have a meaningful graphic representation as a box; An example of simple class is the specification of a simple pump, as showed in Fig. 1 for textual part and in Fig. 2 for the graphical representation.

class clSimplePump -- class header
visible cp, op -- class interface
temporal domain real -- the temporal domain to be considered in the specification
TD Items -- the declarations of time dependent predicates, variables and functions
predicates cp, op, -- close and open messages
expectedOpen
vars expectedState: {open, closed }
axioms -- the axioms of the specification
vars t: real -- the time independent variables, the only ones to be quantified
open: expectedOpen $\leftrightarrow$ Since_{ie} (not cp , op)
state: expectedOpen $\rightarrow$ expectedState =open $\wedge$
$\neg$ expectedOpen $\rightarrow$ expectedState =closed
end clSimplePump

Fig. 1. The class clSimplePump.

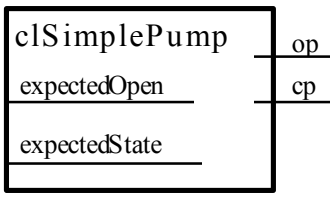
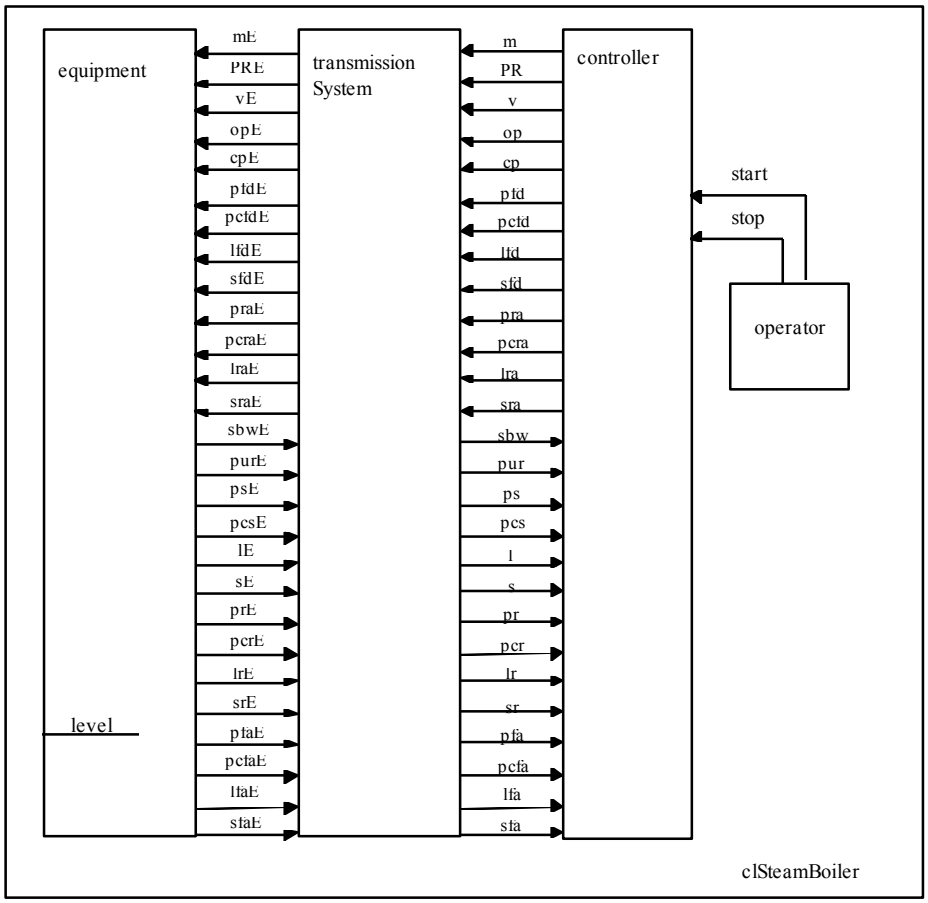


Fig. 2. Graphical representation of class clSimplePump.

The class header is followed by the **visible** clause, which defines the class interface. In the graphic notation the names of the items are written on lines internal to the box; if an item is visible, then the corresponding line continues outside the box. The **axioms** are TRIO formulas and they are prefaced with an implicit universal classical and temporal quantification, i.e., all free variables are universally quantified and an *Always* temporal operator precedes the formula. A name can precede an axiom, to be used as a reference for axiom redefinition in inheritance.

TRIO$^+$ is also endowed with an expressive graphic representation of classes in terms of boxes, arrows, and *connections* to depict class instances and their components, information exchanges and logical equivalence among (parts of) objects. Another TRIO$^+$ facility allows the specifier to describe real world systems that contain groups of identical parts. These configurations are easily described in TRIO$^+$, by defining *arrays* of modules. TRIO$^+$ is also provided with a *genericity* mechanism. Generic classes have one or more parameters that can represent classes of component modules, or scalar constants, or limits of range bounds. Finally, an *inheritance* construct provides the possibility for a class—also named heir class—to receive attributes from other classes.

For the sake of brevity we do not illustrate the features of the above constructs nor exemplify their use: we assume that the parts of specification in the next sections that employ them are sufficiently self explaining, and refer the interest reader to [M&S94].



A.2. Specification of the Steam Boiler

In this appendix we report the complete layout of the module clSteamBoiler in TRIO+.

Then we report the textual declaration of the same class, where for brevity most connection clauses have been omitted, since they already appear explicitly in the graphical class representation.

class clSteamBoiler
temporal domain integer
connections { (transmissionSystem.mE equipment.mE)
...
(controller.m transmissionSystem.m)
...
(operator.start controller.start)

(operator.stop controller.stop) }

TI Items

consts

C: real -- maximal capacity of the steam-boiler

M_1,M_2: real -- minimal and maximal limit of quantity of water

N_1,N_2: real -- minimal and maximal normal quantity of water

W: real -- maximal quantity of steam

U_1,U_2: real -- maximum gradient of increase and decrease

P: real -- nominal throughput of the pumps

nPumps: integer --pumps number

Δ: real -- cycle period in seconds

modules

equipment: clEquipment

transmissionSystem: clTransmissionSystem

operator: clOperator

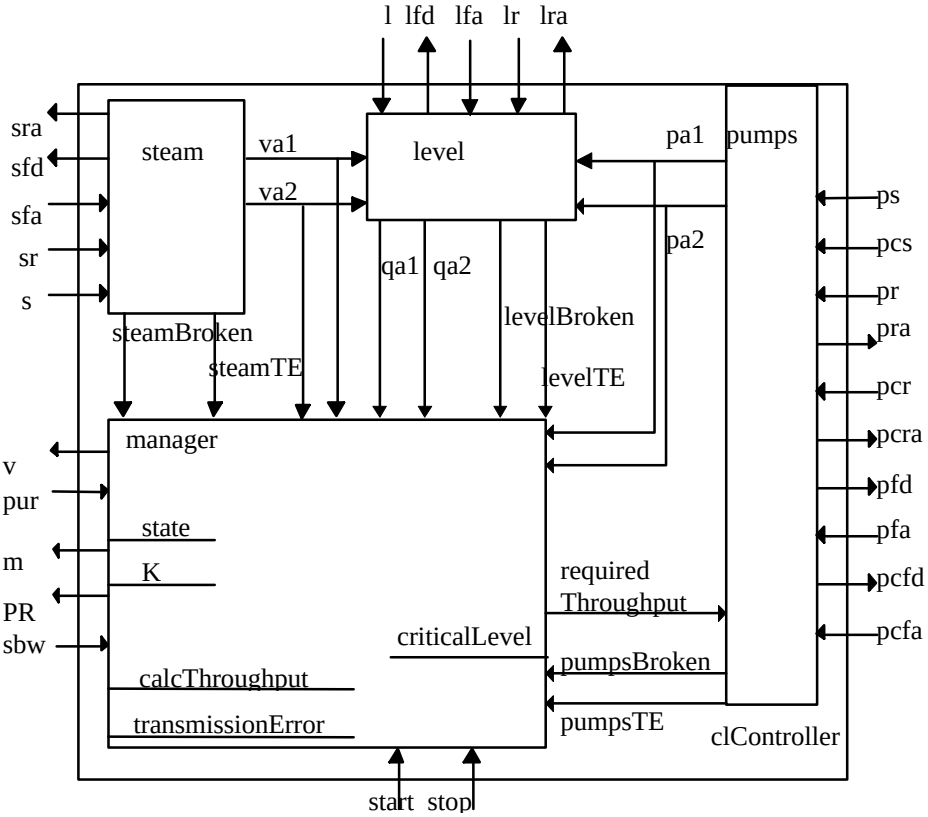
controller: clController[nPumps, C, M_1 , M_2 , N_1 , N_2 , W, U_1 , U_2, P, Δ]

end clSteamBoiler

It can be noticed that the *steamBoiler* class constitutes a closed system, in that there are no externally visible items; the reason for this is that, as anticipated in Section 2.1, the specification models not only the controller to be designed, but also the surrounding environment.

A.3. Controller Module in the Steam Boiler

In this appendix we report the detailed description of controller module. The class, as well as its internal modules, is generic with respect to the parameters of the plant and of the controller program (pumps number, constants C, M_1 , M_2 , N_1 , N_2 , W, U_1 , U_2, P, and cycle period Δ)



class clController [pumpsNumber, C, M_1 , M_2 , N_1 , N_2 , W, U_1 , U_2, P, Δ]

visible -- messages sent by controller

manager.m, manager.PR, manager.v, pumps.op, pumps.cp, -- commands

pumps.pfd, pumps.pcfd, level.lfd, steam.sfd, -- failures detection by controller

pumps.pra, pumps.pcra, level.lra, steam.sra -- acknowledgments

-- messages received by the controller

manager.sbw, manager.pur, -- commands

pumps.ps, pumps.pcs, level.l, steam.s, -- physical units states or measures

pumps.pr, pumps.pcr, level.lr, steam.sr, -- repair messages

pumps.pfa, pumps.pcfa, level.lfa, steam.sfa, -- acknowledgments

manager.start, manager.stop -- from operator

modules

manager: clManager [pumpsNumber, M_1 , M_2 , N_1 , N_2 , U_1 , U_2, W, P, Δ]

steam: clSteam [W, U_1 , U_2, Δ]

level: clLevel [C, U_1 , U_2, Δ]

pumps: clPumps[pumpsNumber, P]

temporal domain integer

connections {

(steam.steamBroken manager.steamBroken) --between manager and steam

...

(level.levelBroken manager.levelBroken) --between manager and level

...

(pumps.pumpsBroken manager.pumpsBroken)--between manager and pumps

...}

end clController

A.4. Level And Steam Module in Controller

class clSteam[W, U_1 , U_2, Δ]

visible s, sr, sfa, sfd, sra, va1, va2, steamTE, steamBroken

temporal domain integer

TD Items

predicates

s(real),	--STEAM(v)
sr,	--STEAM_REPAIRED
sfa,	--STEAM_FAILURE_ACKNOWLEGDEMENT (indicated in [AS] with STEAM_OUTCOME_FAILURE_ACKNOWLEGDEMENT)
sfd,	--STEAM_FAILURE_DETECTION
sra,	--STEAM_REPAIRED_ ACKNOWLEGDEMENT
steamTE, steamBroken	

vars va1, va2: real

axioms

vars

v: real

-- diagnosis: steam is broken iff already broken and not repaired or the measure is out of computed dynamic

diagnosis:

$$\text{steamBroken} \leftrightarrow \left(\begin{array}{l} \text{UpToNow}(\text{steamBroken}) \wedge \neg \text{sr} \vee \\ \text{UpToNow}(\neg \text{steamBroken}) \wedge \forall v \left(s(v) \rightarrow \left(\begin{array}{c} v < 0 \vee v > W \vee \\ v < \text{Past}(va1,1) - U_2 \Delta \vee \\ v > \text{Past}(va2,1) + U_1 \Delta \end{array} \right) \right) \end{array} \right)$$

-- computed maximal and minimal flow of steam if the steam is broken:

calcWithBroken: $\text{steamBroken} \rightarrow \left(\begin{array}{l} va1 = \max(0, \text{Past}(va1,1) - U_2 \Delta) \wedge \\ va2 = \min(W, \text{Past}(va2,1) + U_1 \Delta) \end{array} \right)$

-- if the steam isn't broken va1 and va2 are equal the measured flow:

calc: $\neg$steamBroken $\rightarrow \forall v(s(v) \rightarrow va1=va2=v)$

-- steam failure detection message as soon as steam becomes broken

failureDetection: Becomes(steamBroken) $\rightarrow$Until(sfd,sfa)

-- module sents a sfd message only if a

sfd$\rightarrow$Since($\neg$sfa,Becomes(steamBroken)

-- acknowlegdement of steam repair

repairAck: sra$\leftrightarrow$sr

-- transmission error

transmissionError: $\text{steamTE} \leftrightarrow \left(\begin{array}{c} \neg \exists v \ s(v) \vee \\ \text{sr} \wedge \text{UpToNow}(\neg \text{steamBroken}) \vee \\ \text{sfa} \wedge \text{UpToNow}(\neg \text{sfd}) \end{array} \right)$

end clSteam

class clLevel [C, U_1 , U_2, Δ]

visible l, lfd, lfa, lr, lra, qa1, qa2, levelTE, levelBroken

temporal domain integer

TD Items

predicates

l (real), --LEVEL
lr, --LEVEL_REPAIRED
lfa, --LEVEL_FAILURE_ACKOWLEGDEMENT
lfd, --LEVEL_FAILURE_DETECTION
lra, --LEVEL_REPAIRED_
ACKNOWLEGDEMENT
levelTE, levelBroken

vars

qa1, qa2: real

axioms

vars

v: real

-- diagnosis: level is broken iff already broken and not repaired or the measure is out of computed dynamic

diagnosis: $\text{levelBroken} \leftrightarrow \left(\begin{array}{c} \text{UpToNow}(\text{levelBroken}) \wedge \neg \text{lr} \\ \vee \\ \text{UpToNow}(\neg \text{levelBroken}) \wedge \\ \forall v \left(l(v) \rightarrow \left(\begin{array}{c} v < 0 \vee v > C \vee \\ v < \text{Past}(qa1,1) - va2\Delta - \frac{1}{2} U_1 \Delta^2 + pa1\Delta \vee \\ v > \text{Past}(qa2,1) - va1\Delta + \frac{1}{2} U_2 \Delta^2 + pa2\Delta \end{array} \right) \right) \end{array} \right)$

-- computed maximal and minimal quantity of water if the level is broken:

calcWithBroken:

$$\text{levelBroken} \rightarrow \left(\begin{array}{l} qa1 = \max(0, \text{Past}(qa1,1) - va2\Delta - \frac{1}{2} U_1 \Delta^2 + pa1) \wedge \\ qa2 = \min(C, \text{Past}(qa2,1) - va1\Delta + \frac{1}{2} U_2 \Delta^2 + pa2) \end{array} \right)$$

-- if the level isn't broken qa1 and qa2 are equal the measured quantity:

calc: $\neg$levelBroken $\rightarrow \forall v(l(v) \rightarrow qa1=qa2=v)$

-- level failure detection message as soon as level becomes broken

failureDetection: (Becomes(levelBroken) $\rightarrow$Until(lfd,lfa))
$\wedge$ (lfd$\rightarrow$Since($\neg$lfa,Becomes(levelBroken))

-- acknowlegdement of level repair

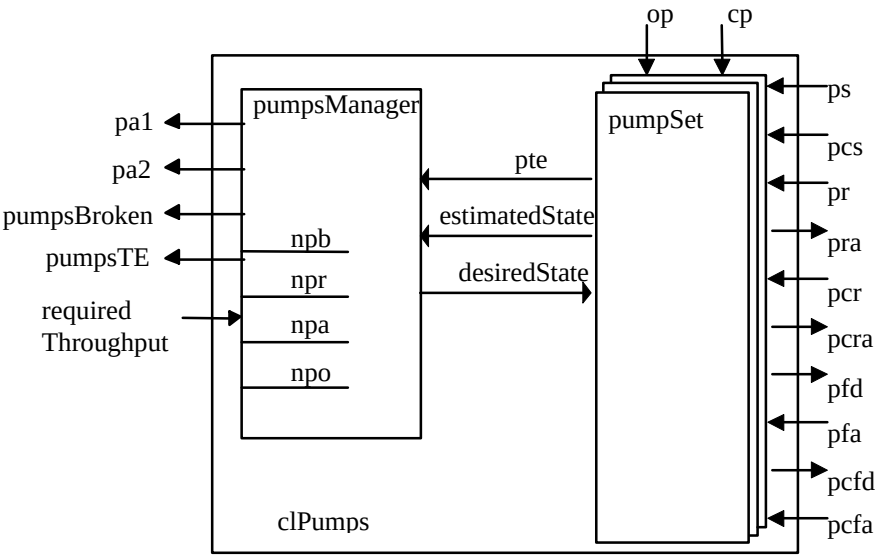
repairAck: lra$\leftrightarrow$lr

-- transmission error

transmissionError: $\text{levelTE} \leftrightarrow \left(\begin{array}{c} \neg \exists v l(v) \vee \\ \text{lr} \wedge \text{UpToNow}(\neg \text{levelBroken}) \vee \\ \text{lfa} \wedge \text{UpToNow}(\neg \text{lfd}) \end{array} \right)$

end clLevel

A.5. Pumps Module in Controller



class clPumps [pumpsNumber, P]
visible pa1,pa2, pumpsTE, pumpsBroken, requiredThroughput, op, cp, ps, pcs, pr, pra, pcr, pcra, pfd, pcfd, pfa, pcfa
modules pumpSet: **array** [1..pumpsNumber] **of** clPump
pumpsManger: clPumpsManager [pumpsNumber, P]
temporal domain integer
connections {
(pumpsManager.pa1 pa1) --from or to pumpsManager
...
(op.pumpSet op) -- from ot to pumpSet
(cp.pumpSet cp)
...
(pumpSet .pte pumpsManager.pte)--between pumpSet and pumpsManager
...}
TD Items
predicates
op(1..pumpsNumber), --OPEN_PUMP(i)
cp(1..pumpsNumber), --CLOSE_PUMP(i)
ps(1..pumpsNumber, {open,closed}), --PUMP_STATE(n,b)
pcs(1..pumpsNumber, {open,closed}), --PUMP_CONTROL_STATE(n,b)
pr(1..pumpsNumber), --PUMP_REPAIRED
pra(1..pumpsNumber), -- PUMP_REPAIRED_ACKNOWLEDGEMENT
pcr(1..pumpsNumber), -- PUMP_CONTROL_REPAIRED

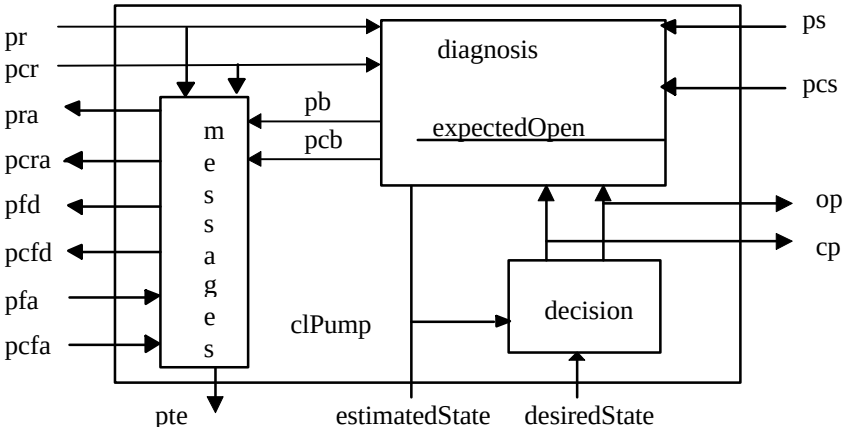
pcra(1..pumpsNumber), --
PUMP_CONTROL_REPAIRED_ACKNOWLEDGEMENT
pfd(1..pumpsNumber), -- PUMP_FAILURE_DETECTION
pcfd(1..pumpsNumber), -- PUMP_CONTROL_FAILURE_DETECTION
pfa(1..pumpsNumber), -- PUMP_FAILURE_ACKNOWLEDGEMENT
pcfa(1..pumpsNumber), --
PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT
pumpsTE, -- transmission error detection
pumpsBroken -- at least a pump is broken

vars

requiredThroughput: positive real --
pa1: positive real-- minimal adjusted total throughput of pumps
pa2: positive real-- maximal adjusted total throughput of pumps

end clPumps

A.6. Single Pump Module in Pumps Module



class clPump

visible ps, pcs, pr, pcr, pra, pcra, pfd, pcfd,
pfa, pcfa, op, cp, estimatedState, desiredState, pte

modules messages:clPumpMessagges
decision: clDecision
diagnosis: clDiagnosis

connections {
(ps diagnosis.ps) --from or to diagnosis
...
(pr messages.pr) --from or to messages
...
(decison.op op) --from or to decision

...
(diagnosis.pb messages.pb) -- between messages and diagnosis
...
(decision.op diagnosis.op) --between decision and diagnosis
...
}

temporal domain integer

TD Items

predicates

op, cp, ps({open,closed}), pcs({open,closed}), pr, pra, pcr, pcra, pfd, pcfd, pfa, pcfa, pte

vars

desiredState: {open, closed}

estimatedState: {open, closed, broken}

end clPump

A.7. Diagnosis Module in Pump Module

Module *diagnosis* specifies how faults of the pump or of its controller are determined.

class clDiagnosis

visible ps, pcs, pr, pcr, op, cp, pb, pcb, estimatedState

TD Items

predicates expectedOpen --true if the pump would be open
ps ({open, closed}) --state of the pump in input
pcs({open, closed}) --state of the pump control in input
pr --message indicating that the pump has been repaired
pcr --message indicating that the pump control has been repaired
op -- the pump receives a open command
cp -- the pump receives a close command
pb -- diagnosis estimates the pump broken
pcb -- diagnosis estimates the pump control broken
estimatedState({open,closed,broken}) --the diagnosed state

axioms

vars

-- the pump would be open iff a open command is not followed by any close command

open: expectedOpen $\leftrightarrow$ $Since_{ie}(\neg cp, op)$

--the pump is broken iff it was broken and now it isn't repaired or it is in a state different from the expected one. Idem for the pump control.

pumpDiagnosis: $$pb \leftrightarrow \begin{pmatrix} UpToNow(pb) \wedge \neg pr \quad \vee \\ expectedOpen \wedge ps(closed) \quad \vee \\ \neg expectedOpen \wedge ps(open) \end{pmatrix}$$

pumpControlDiagnosis: $\text{pcb} \leftrightarrow \begin{pmatrix} \text{UpToNow}(\text{pcb}) \wedge \neg\text{pcr} \vee \\ \text{expectedOpen} \wedge \text{pcs}(\text{closed}) \vee \\ \neg\text{expectedOpen} \wedge \text{pcs}(\text{open}) \end{pmatrix}$

estimatedBroken: $\text{pb} \wedge \text{pcb} \rightarrow \text{estimatedState}=\text{broken}$

estimatedState:

$$\left(\neg\text{pb} \wedge \text{pcb} \rightarrow \begin{pmatrix} (\text{ps}(\text{open}) \rightarrow \text{estimatedState} = \text{open}) \wedge \\ (\text{ps}(\text{closed}) \rightarrow \text{estimatedState} = \text{closed}) \end{pmatrix}\right)$$

$$\wedge$$

$$\left(\begin{pmatrix} \neg\text{pb} \wedge \neg\text{pcb} \vee \\ \text{pb} \wedge \neg\text{pcb} \end{pmatrix} \rightarrow \begin{pmatrix} (\text{pcs}(\text{open}) \rightarrow \text{estimatedState} = \text{open}) \wedge \\ (\text{pcs}(\text{closed}) \rightarrow \text{estimatedState} = \text{closed}) \end{pmatrix}\right)$$

end clDiagnosis

The following class *clDiagnosisWithCoherenceControl* is a first elaboration on the diagnosis criteria, and is based on the idea of confronting the state of the pump and of its controller: a difference between such states is considered a symptom of a fault. Notice that class *clDiagnosisWithCoherenceControl* is defined as a heir of class *clDiagnosis*, so that it can share with it the interface and the local and exported items: its definition requires, in practice, only the redefinition of the axioms for the diagnosis. Class *clDiagnosisWithCoherenceControl* can be used to define a new class, call it *clPumpWithCoherenceControl*, as an heir of *clPump* where the diagnosis module (originally defined of class *clDiagnosis*) is redefined to be of the class *clDiagnosisWithCoherenceControl*.

class clDiagnosisWithCoherenceControl

inherit clDiagnosis [**redefine** pumpDiagnosis, pumpControlDiagnosis]

axioms

pumpDiagnosis: $\text{pb} \leftrightarrow \begin{pmatrix} \text{UpToNow}(\text{pb}) \wedge \neg\text{pr} \vee \\ \text{expectedOpen} \wedge \text{ps}(\text{closed}) \vee \\ \neg\text{expectedOpen} \wedge \text{ps}(\text{open}) \vee \\ \begin{pmatrix} \text{UpToNow}(\neg\text{pcb}) \vee \\ \text{pcr} \end{pmatrix} \wedge \begin{pmatrix} \text{ps}(\text{closed}) \wedge \text{pcs}(\text{open}) \vee \\ \text{ps}(\text{open}) \wedge \text{pcs}(\text{open}) \end{pmatrix} \end{pmatrix}$

pumpControlDiagnosis:

$$\text{pcb} \leftrightarrow \begin{pmatrix} \text{UpToNow}(\text{pcb}) \wedge \neg\text{pcr} \vee \\ \text{expectedOpen} \wedge \text{pcs}(\text{closed}) \vee \\ \neg\text{expectedOpen} \wedge \text{pcs}(\text{open}) \vee \\ \begin{pmatrix} \text{UpToNow}(\neg\text{pb}) \vee \\ \text{pr} \end{pmatrix} \wedge \begin{pmatrix} \text{ps}(\text{closed}) \wedge \text{pcs}(\text{open}) \vee \\ \text{ps}(\text{open}) \wedge \text{pcs}(\text{open}) \end{pmatrix} \end{pmatrix}$$

--the pump is broken iff it was broken and now it isn't repaired or it is in a state different from the expected one or furthermore pump and pump control are in different states. The pump state is compared with pump control state only if pump control was not broken or now repaired. Idem for the pump control.

end clDiagnosisWithCoherenceControl

The following class *clDiagnosisWithAsymmetricControl* is a further, last elaboration on the diagnosis criteria where it is considered highly unlikely that both the pump and its controller are simultaneously broken and therefore, in case they give unexpected but mutually consistent indication, it is assumed that only the pump is broken.

class clDiagnosisWithAsymmetricControl
inherit clDiagnosisWithCoerenceControl [**redefine** pumpControlDiagnosis]
axioms
pumpControlDiagnosis:

$$\text{pcb} \leftrightarrow \left(\begin{array}{c} \text{UpToNow}(\text{pcb}) \wedge \neg \text{pcr} \vee \\ \left(\begin{array}{c} \text{UpToNow}(\neg \text{pb}) \vee \\ \text{pr} \end{array} \right) \wedge \left(\begin{array}{c} \text{ps}(\text{closed}) \wedge \text{pcs}(\text{open}) \vee \\ \text{ps}(\text{open}) \wedge \text{pcs}(\text{open}) \end{array} \right) \end{array} \right)$$

--the pump control is broken iff it was broken and now it isn't repaired or it is in a state different from the pump state. If pump control and pump states are equal, but different from the expected state, diagnosis esteems broken only the pump.

end clDiagnosisWithAsymmetricControl

Notice how the inheritance construct can be used to describe easily any policy in fault detection reusing as much as possible from previous definitions, and composing new definitions with preceding ones in a very simple manner.

A.8. Decision Module in Pump Module

The class *clDecision* reported below simply determines the actual commands to be sent to the pumps based on its desired state (as decided by the *pumpManager* module) and its current state (as determined by the *diagnosis* module).

class clDecision
visible op, cp, estimatedState, desiredState
temporal domain integer
TD Items
predicates
op -- decision sends a open command
cp -- decision sends a close command
vars
desiredState: {open, closed}
estimatedState: {open, closed, broken}
axioms
closePump:

$$\left(\begin{array}{c} \text{Becomes}(\text{estimatedState} = \text{broken}) \\ \vee \\ \text{estimatedState} = \text{open} \wedge \text{desiredState} = \text{closed} \end{array} \right) \leftrightarrow \text{cp}$$

openPump: estimatedState=closed $\wedge$ desiredState=open $\leftrightarrow$ op
end clDecision

A.9. Pump Messages Module in Pump Module

The class *clPumpMessages* simply manages the exchange with the equipment of information regarding the faults of a pump and of its controller.

class clPumpMessages
visible pr, pcr, pra, pcra, pfd, pcfd, pfa, pcfa, pte, pb, pcb
temporal domain integer
TD Items
predicates
pr, pcr, pra, pcra, pfd, pcfd, pfa, pcfa, pte, pb, pcb
axioms
vars
s: {open, closed}
-- pump failure detection message as soon as pump becomes broken
module sents a sfd message only if a failure was detected

$$(\text{Becomes}(\text{pb}) \rightarrow \text{Until}(\text{pfd}, \text{pfa})) \wedge (\text{pfd} \rightarrow \text{Since}(\neg\text{pfa}, \text{Becomes}(\text{pb})))$$

pumpFailureDetection:

-- pump control failure detection
pumpControlFailureDetection: (Becomes(pcb) →Until(pcfd,pcfa))
∧ (pcfd→Since(¬pcfa,Becomes(pcb))
-- acknowlegdement of pump repair
pumpRepairAck: pra↔pr
-- acknowlegdement of pump control repair
pumpControlRepairAck: pcra↔pcr
-- transmission error

$$\textit{transmissionError}:\ \text{pte} \leftrightarrow \begin{pmatrix} \neg\exists s\ \text{ps}(s) \vee \neg\exists s\ \text{pcs}(s) \vee \\ \text{pr} \wedge \text{UpToNow}(\neg\text{pb}) \vee \\ \text{pcr} \wedge \text{UpToNow}(\neg\text{pcb}) \vee \\ \text{pfa} \wedge \text{UpToNow}(\neg\text{pfd}) \vee \\ \text{pcfa} \wedge \text{UpToNow}(\neg\text{pcfd}) \end{pmatrix}$$

end clPumpMessages

A.10. Pump Manager in Pump Module

Finally, *clPumpManager*, the class of module *pumpManager* in the class clPumps (see Fig. 5) formalizes the very general and abstract requirement that at any time a number of pumps related to the requiredThroughput must be open, without indicating any particular policy. As in the case of pump diagnosis, particular criteria or strategies in alternating pumps can be specified by means of the inheritance construct (we do not further develop our exercise in this direction for the sake of brevity).

class clPumpsManager[pumpsNumber, P]

visible pa1, pa2, pumpsTE, pumpsBroken, requiredThroughput, pte, estimatedState, desiredState

temporal domain integer

TD Items

predicates

pumpsTE, -- transmission error detection

pumpsBroken -- at least a pump is broken

pte (1..pumpsNumber) -- pte(i) iff i-st pump in pumpset detects a transmission error

vars

pa1: positive real

pa2: positive real

requiredThroughput: positive real

npb: [0..pumpsNumber] -- number of broken pumps

npr: [0..pumpsNumber] -- number of required pumps

npa: [0..pumpsNumber] -- number of avaible pumps

npo: [0..pumpsNumber] -- number of open pumps

functions

-- estimated states for every pump in pumpSet from pump modules

estimetedState (1..pumpsNumber) : {open,closed,broken}

-- desired states for every pump in pumpSet in accordance with required throughput

desiredState (1..pumpsNumber) : {open,closed}

axioms

vars

i,x: 1..pumpsNumber

diagnosis: pumpsBroken $\leftrightarrow \exists$i (estimatedState(i)=broken)

transmissionError: pumpsTE$\leftrightarrow \exists$i pte(i)

claculation: pa1=npo*P $\wedge$ pa2=(npo+npb)*P

numberBroken: npb=x $\leftrightarrow \exists_x$ i (estimatedState(i)=broken)

numberOpen: npo=x $\leftrightarrow \exists_x$ i (estimatedState(i)=open)

numberAvaible: npa= pumpsNumber -npb

numberRequired: npr=requiredThroughput DIV P

-- if the number of required pumps is greater than the number of avaible pumps, every pump (not broken) would be open; otherwise exactly npr pumps would be open

desiderata:

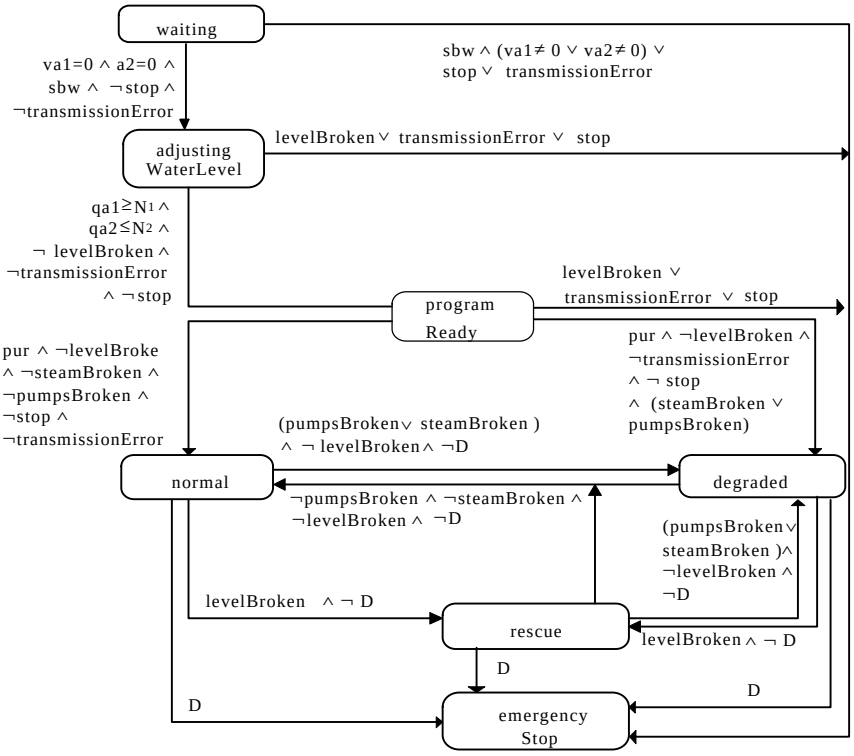
npr$\geq$npa $\rightarrow \forall$i (estimatedState(i) $\neq$ broken $\rightarrow$ desiredState(i)=open) $\wedge$

npr<npa $\rightarrow \exists_{npr}$ i (estimatedState(i) $\neq$ broken $\wedge$ desiredState(i)=open)

end clPumpsManager

A.11. Manager Module in Controller

The behavior of the Manager component can be modeled by the finite automaton reported below. This automaton can be translated into a set of TRIO axioms; an example of this transaction is reported in the class clManager



D ≡ criticalLevel ∨ stop ∨ transmissionError

class clManager[pumpsNumber, M_1 , M_2 , N_1 , N_2 , U_1 , U_2, W, P, Δ]

visible stop, start, v, pur, m, PR, sbw, steamBroken, steamTE, va1, va2, levelBroken, levelTE, qa1, qa2, pumpsBroken, pumpsTE, pa1, pa2, requiredThroughtput

temporal domain integer

TD Items

predicates

m({initialization, normal, degraded, rescue, emergencyStop}), --MODE
PR, --PROGRAM_READY
v, --VALVE
pur, -- PHYSICAL_UNITS_READY
sbw, -- STEAM_BOILER_WAITING
start, -- start request by operator

stop -- STOP message by operator
transmissionError -- a transmission failure is dectected
steamTE -- a transmission failure in steam module
levelTE -- a transmission failure in level module
pumpsTE -- a transmission failure in pumps module
steamBroken -- steam is broken
levelBroken
pumpsBroken -- at least a pump is broken
critcalLevel -- the calculated level of water is out of safety range

vars

state { waiting, adjustingWaterLevel, programReady, normal, degraded, rescue, emergencyStop} -- actual state
calcThroughput : real -- calculated throghput
requiredThroughput : positive real -- required throghput to pumps
K --safety timeout for rescue mode
va1, va2: real --minimal and maximal adjusted flow of exiting steam
pa1, pa2: real --minimal and maximal adjusted throghput of the pumps
qa1, qa2: real --minimal and maximal adjusted quantity of water

axioms

vars

initModeFromState:
m(initialization) ↔ state=waiting ∨ state=adjustingWaterLevel ∨ state=programReady
modeFromState: ¬m(initialization) → m(state)
-- in initializing mode system state can be waiting or adjustingWaterLevel or programReady. In the other modes state and mode are identical
transmissionErrorDetection:
transmissionError ↔ steamTE ∨ levelTE ∨ pumpsTE
calculation:

$$\text{calcThroughput} = \frac{\left(\frac{N_1 + N_2}{2} - \frac{qa1 + qa2}{2}\right) \cdot \text{pumpsNumber} \cdot P}{\frac{N_1 + N_2}{2} - M_1} + \frac{va1 + va2}{2}$$

request: calcThroughput <0 → requiredThroughput=0 ∧
calcThroughput ≥0 → requiredThroughput= calcThroughput
valveManagement:

$$\begin{pmatrix} \text{Becomes}(\text{state} = \text{adjustingWaterLevel}) \wedge qa1 > N_2 \vee \\ \text{state} = \text{adjustingWaterLevel} \wedge \text{Becomes}(qa1 \le N_2) \end{pmatrix} \leftrightarrow v$$

programReadyComunication: PR↔state=programReady

KCalculation:

$$\left(\begin{array}{c}\text{Becomes}(\text{levelBroken})\wedge\\ x=\min\left(\dfrac{M2-\text{Past}(qa2,1)}{\text{pumpsNumber}\cdot P},\dfrac{\text{Past}(qa1,1)-M1}{W}\right)\end{array}\right)\rightarrow \text{Until}\left(\begin{array}{l}K=x,\\ \text{Becomes}(\text{levelBroken})\end{array}\right)$$

criticalLevelDetection:

$$\text{criticalLevel}\leftrightarrow\left(\begin{array}{c}\text{mode}(\text{rescue})\wedge \text{Lasted}_{ii}(\text{mode}(\text{rescue}),K)\vee\\ \neg\text{mode}(\text{rescue})\wedge\left(\begin{array}{c}\text{Past}(qa1,1)-va2D-\frac{1}{2}U_1D^2+pa1D\le M_1\vee\\ \text{Past}(qa2,1)-va1D+\frac{1}{2}U_2D^2+pa2D\ge M_2\end{array}\right)\end{array}\right)$$

normalToRescue

$$\left(\begin{array}{c}\text{UpToNow}(\text{mode}(\text{normal}))\wedge\\ \text{levelBroken}\wedge\\ \neg D\end{array}\right)\rightarrow\left(\begin{array}{c}\text{mode}(\text{rescue})\wedge\\ \text{Until}\left(\text{mode}(\text{rescue}),\neg\text{levelBroken}\vee D\right)\end{array}\right)$$

-- similar rules can be derived systematically for alla other transitions.

end clManager