Towards a Methodological Approach to Specification and Analysis of Dependable Automation Systems

Simona Bernardi[1], Susanna Donatelli[1], and Giovanna Dondossola[2] ⋆

[1] Dipartimento di Informatica, Università di Torino, Italy, {susi,bernardi}@di.unito.it
[2] CESI Automation & Information Technology, Milano, Italy, dondossola@cesi.it

Abstract. The paper discusses a constructive approach to the temporal logic specification and analysis of dependability requirements of automation systems. The work is based on TRIO formal method, which supports a declarative temporal logic language with a linear notion of time, and makes use of UML class diagrams to describe the automation system. The *general* concepts presented for the automation system *domain* are here *instantiated* on a case study *application* taken from the energy distribution field.

1 Introduction

The design of critical systems is faced with the need of devising appropriate "dependability strategies", that is to say the need of choosing and specifying a set of steps that allow to improve the reliability of the system. In the project DepAuDE [6][3] this issue has been investigated and a methodology to support the analyst in collecting and analyzing system dependability requirements, aimed at designing appropriate Fault Tolerance (FT) solutions, has been devised through a collaboration between the CESI [4] company and the University of Torino.
The application *domain* for the methodology is that of distributed cyclic control systems, while the specific *application* considered in the case study presented here is related to the automation system for primary substations of electricity distribution network (called PSAS in the following), proposed by CESI within the DepAuDE project [5]. The PSAS provides the tele-control and protection functions of the Primary Sub-stations (PSs), where PSs are nodes of the electric distribution grid connecting the High Voltage transportation network to the Medium Voltage distribution. The aspects of PSAS that are relevant for this paper concern the cyclic behavior and the synchronization issues of the distributed automation systems local to the PS and they will be introduced, when needed, in Section 4.

Three different formalisms collaborate in a synergic manner in the methodology: UML Class Diagrams [21] (CDs from now on), a static paradigm, TRIO [12] temporal

⋆ Partially funded by Italian Ministry of Productive activities - Rete 21 - SITAR project

[3] EEC-IST-2000-25434 DepAuDE (Dependability for embedded Automation systems in Dynamic Environment with intra-site and inter-site distribution aspects) project.

[4] CESI is an Italian company providing services and performing research activities for the Electric Power System.

logic, and Stochastic Petri nets [19] (PN), an operational paradigm aimed at performance and dependability evaluation. A multi-formalism approach during the dependability process is also advocated by emerging standards like IEC 60300 [4].

Class Diagrams are the "entry level" in the methodology that provides a set of predefined CDs for the automation system domain, called "the *generic* scheme", and guidelines on how to produce from it an *instantiated* one, that refers to the target application. Diagrams are meant as a support for the requirements collection and/or for structuring and/or reviewing for completeness already available requirements. In DepAuDE we have studied how the information available in the scheme can be used as a starting point for the modelling efforts with TRIO and SPN. The role of Stochastic Petri nets in the DepAuDE approach is to evaluate the reliability of the candidate dependability strategies [7, 1], while in this paper we discuss the role of TRIO and its links to the CDs.

TRIO is a linear temporal logic that finds its origins in the nineties as a joint effort of Politecnico di Milano and ENEL as a formal declarative language for real-time systems. Since then several TRIO dialects and validation tools have been prototyped (e.g., [18], [10]) and used in several projects. We use Modular TRIO Language [3] and a tool set developed in the FAST project[5], implemented over the Prover Kernel [9].

In DepAuDE TRIO has been used *for specifying and analysing dependability requirements and fault tolerance strategies in a temporal logic framework*. The choice of a declarative language, and in particular a logic formalism like TRIO, has been driven by the analysis methods followed at CESI, where TRIO is a common practise for system analysis of timed properties. Other choices are indeed possible, like that of using an operational formalism using extended State-charts, as proposed, for example in the embedded system field, in [14]: the advantages and disadvantages of operational versus declarative formalisms are well established, and we shall not discuss them here.

The work on CDs and TRIO in DepAuDE takes its basis from the preliminary work in [11], in which the first ideas on the use of CD in the context of dependability analysis of automation systems and the possibilities of cooperation of TRIO specification with CD models were discussed. The work presented in this paper represents a step forward, by introducing a three steps incremental specification: the *derivation* of the TRIO specification structure from the UML Class Diagrams, a *first completion* of the specification with domain dependent knowledge and the *full formalisation* with application dependent knowledge. Goal of this three steps procedure is to provide the user with a more structured approach to the construction of logic specifications, and to allow reuse of partial specifications.

This paper describes the three steps and demonstrate their efficacy through the PSAS case study. Due to space constraints, the paper concentrates the analysis only on timing properties, while the complete case study can be found in [7].

The paper is structured as follows. Section 2 recalls the language TRIO and its analysis capability, Sect. 3 summarizes the CD scheme proposed in DepAuDE, Sect. 4 introduces the three steps procedure and its application to the PSAS, while Sect. 5 discuss the analysis methodology with examples from the PSAS.

[5] ESPRIT FAST Project No. 25581 (Integrating Formal Approaches to Specification, Test case generation and automatic design verification)

2 Basic TRIO Methodology

A TRIO specification is structured into classes, and each class includes a declaration session followed by a formulae session. The declaration session defines the signature of TRIO items (atomic propositions, predicates, values and functions), which are grouped into Time Independent (TI) and Time Dependent (TD) items, and the types for the value domains of predicates, values and functions.

TRIO formulae are expressed in a temporal logic language that supports a linear notion of *discrete* time. Beyond the *propositional operators* and, or, xor, implies, iff ($\&$, $|$, $||$, $\rightarrow$, $\leftrightarrow$ in TRIO syntax) and the *quantifiers* $\exists, \forall, \nexists$ (all, ex, nex in TRIO), TRIO formulae can be composed using the *primitive* temporal operator *Dist*, and *derived* temporal operators. *Dist* allows to refer to events occurring in the future or in the past with respect to the current, implicit time instant. If F is a TRIO formula and δ is a term of time type, then $Dist(F, \delta)$ is satisfied at the current time instant if and only if F holds at the instant laying δ time units ahead (or behind if t is negative) the current one. Derived temporal operators can be defined from *Dist* through propositional composition and first order quantification on variables representing a time distance [3]. The intuitive semantic of the operators used in this paper is as follows: $\mathbf{Alw}(F)$ (F is always true), $\mathbf{AlwF}(F)$ (F will be always true in the future), $\mathbf{AlwP}(F)$ (F has been always true in the past), $\mathbf{Becomes}(F)$ (F is true now and it was false in the instant immediately preceding the current one), $\mathbf{NextTime}(F, \delta)$ (F will become true exactly at δ time and from now till that instant it will be false). TRIO is linear and time is implicit: all properties refer to a single execution observed at the current time.

The formulae session may include: *definitions, axioms, properties* and *Abstract Test Cases (ATC)*: they are all temporal logic formulae, but they play a different role in the analysis. *Definitions* are a macro-expansion mechanism, *axioms* express system requirements (the description of the system), *properties* express requirements that have to be derivable from the set of axioms (the system properties of interest), and *ATC* are formulae compatible with the axioms that are used to focus the analysis on relevant, more restricted, contexts (a particular behavior of the system).

The TRIO tool supports automatic proof sessions based on three proof techniques: *model generation, property proof*, and *test case generation*. Model generation produces a set of temporal logic models (called TRIO histories) for the selected properties: a model is graphically represented by a set of up/down functions plotting the truth value of a Time Dependent (TD) proposition/predicate on the time line. Property proof computes the validity of a property. If the property is not valid then counter models are produced. Test case generation allows the automatic generation of a set of test cases according to a number of testing criteria. The analysis requires the setup of the "proof session" to specify the portion of the specification to be used for the proof, the choice of the proof technique, and the setting of the time interval considered.

3 UML Class Diagrams for Automation Systems

The DepAuDE "generic scheme" consists of a set of UML Class Diagrams (CDs) capturing generic issues considered relevant for a wide class of dependable automation applications. From the generic scheme, an instantiation activity (described in [2]) allows to derive an instantiated CD scheme, that specifies a given application (system description and associated dependability requirements).

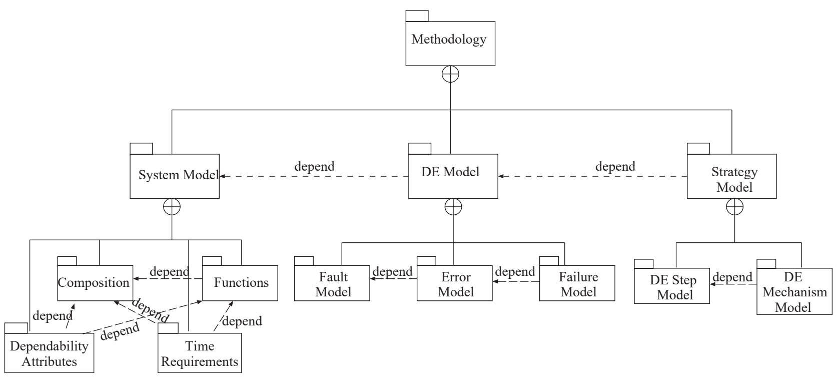
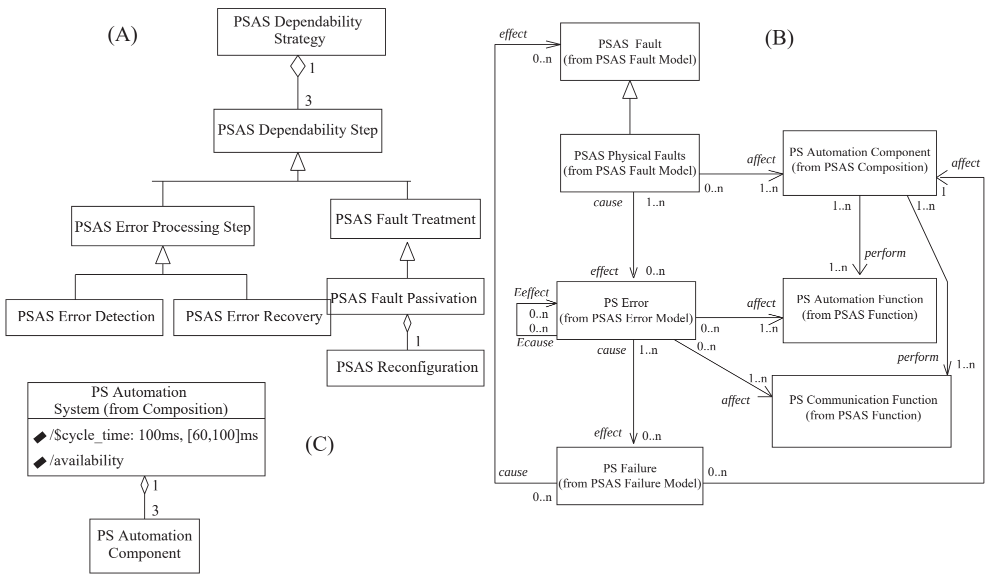


Fig. 1. Hierarchical Structure of the packages.

The set of CDs are grouped into the hierarchical structure of UML *packages* represented in Fig. 1, where each non-leaf package encapsulates a set of inner packages together with their dependency relationships, that indicates a suggested order of use. For each innermost package one or more CDs are provided that constitute different *views* on the system being described, focusing on aggregation, generalization/specialization, and class definition (associations and attributes) aspects of a portion of the system. In the scheme the class attributes are stereotyped to represent either parameters provided as input to the specifications or measures to be evaluated or upper/lower bounds to be validated. Let us now provide an overview of the scheme, following Fig. 1.

System Model addresses the system requirements. It specifies 1) the conceptual structure of an automation system ; 2) the association of automation functions to system components; 3) the association of (real) time requirements to system components and/or functions; and, finally, 4) the association of dependability attributes to system components and/or functions. The CDs that are most relevant for the presented case study are the CD *Structure* in which the whole automated system is decomposed into automation sites connected by an automation communication infrastructure. An automation system residing on a given site is defined as an aggregation of automation components and of automation functions. An automation component may control directly a (set of) plant components through association *control*. The CD *Constraints* allows to identify those temporal attributes which are considered relevant for the specification of automation systems, such as *cycle_time* that refers to the time required by the automation system to execute a complete cycle (i.e., read a sample input from the plant, elaborate to produce the future state and provide output to the plant). The attribute *cycle_time* is defined as a bound to be validated in the generic CD, and as a specific value (100ms) on the instantiated CD.

Figure 2(C) shows a very small portion of the CD that describes the PSAS automation system, made of three Automation Components and with two relevant attributes.



(A)
PSAS Dependability Strategy
1
3
PSAS Dependability Step
PSAS Error Processing Step
PSAS Fault Treatment
PSAS Error Detection
PSAS Error Recovery
PSAS Fault Passivation
1
PSAS Reconfiguration
PS Automation System (from Composition)
/$cycle_time: 100ms, [60,100]ms
/availability
(C)
1
3
PS Automation Component
(B)
effect
0..n
PSAS Fault (from PSAS Fault Model)
PSAS Physical Faults (from PSAS Fault Model)
affect
0..n
1..n
PS Automation Component (from PSAS Composition)
affect
1
cause
1..n
1..n
1..n
perform
1..n
effect
0..n
Eeffect
0..n
0..n
Ecause
PS Error (from PSAS Error Model)
affect
0..n
1..n
PS Automation Function (from PSAS Function)
cause
1..n
0..n
perform
1..n
1..n
affect
PS Communication Function (from PSAS Function)
effect
0..n
cause
0..n
PS Failure (from PSAS Failure Model)
0..n

Fig. 2. View of the instantiated CD *Strategy* (A) *FEF chain* (B), and *Strategy* (C).

Dynamic Environment Model. The package *DE Model* captures several concepts on the fault theory expressed in the literature [16], and its extension to malicious faults, as developed by the European MAFTIA project [24] and partially modified upon CESI experience. It contains three sub-packages (*Fault Model*, *Error Model* and *Failure Model*) each one characterizing a different view of a fault evolution, from its appearance to its recovery and/or repair. The CDs are connected so as to reflect the propagation effect linking faults to errors and errors to failures (FEF chain).

Once customized on a specific application, the CD of the FEF chain shows which faults provoke which errors and which (set of) errors provoke a failure. The diagram also connects each type of fault, error and failure with the corresponding system components affected by it: a fault may affect an automation component (elaboration, memory or communication unit), and an error may affect an automation function performed by the faulty component. If a function is affected by an error, the error can be propagated to another function thus provoking an error in another function. If errors are not recovered in due time failures may appear. The FEF chain for the PSAS is given in Fig. 2(B).

Strategy Model. This package concerns the representation of the dependability strategy. A dependability strategy is defined as an aggregation of (temporal) steps in which actions have to be undertaken in order to stop the fault evolution. The *Dependability Step* CD supports a classification of those steps and connects them to the fault, error, and failure elements addressed by the step. The PSAS strategy consists of three steps:

an error detection followed by an attempt of error recovery and, eventually, a system reconfiguration: the correspondent CD is shown in Fig. 2(A).

4 TRIO Scheme in DepAuDE

The TRIO formalization is aimed at describing and analyzing the logic of a dependability strategy following the requirements collected and structured according to the UML scheme. The analysis concerns the temporal evolution of an automation system integrating a dependability strategy. The TRIO specification is built incrementally, and each partial specification is validated by several proof sessions.

In the DepAuDE Methodology the TRIO formalisation of dependability requirements is an extension of their representation in UML Class Diagrams. The relation between UML class diagrams and TRIO forms is a partial one: only a subset of the UML class attributes and associations is related with elements of the TRIO scheme and, vice-versa, which is not surprising since the two formalisms play quite different roles in the development of a system. In particular, TRIO classes introduce new time relationships, which are not present in the correspondent CDs.

The approach used to develop a TRIO specification is a *three-steps* procedure. The first step consists of deriving a basic TRIO specification structure, using a set of predefined actions that are applied using information from the instantiated UML scheme (Sect. 4.1). In the second step, domain specific knowledge is introduced leading to partially defined classes that include item declarations and formulae of general usage (Sect. 4.2). In the third step the specification is completed using application dependent knowledge and design level information (Sect. 4.3).

4.1 Deriving the Skeleton of the TRIO Scheme from UML Class Diagrams

As a starting point a number of syntactic links have been identified between CDs elements and TRIO elements:

L1 Reuse of the structured organization into classes;
L2 The objects instances of UML classes are mapped into TRIO types;
L3 Class attributes are mapped into TRIO time (in)dependent items;
L4 The value of a class attribute is mapped into a TRIO axiom, if the value is unique, or into a TRIO type, otherwise;
L5 Associations are mapped into TRIO time (in)dependent predicates and axioms;
L6 UML constraints are mapped into TRIO properties.

Each first level package of the UML Dependability Scheme given in Fig. 1 maps to a TRIO class, leading to a TRIO Scheme with three classes: *System, Dynamic Environment* and *Strategy*. In this paper we only provide a partial derivation for the three classes: their full description can be found in [7].

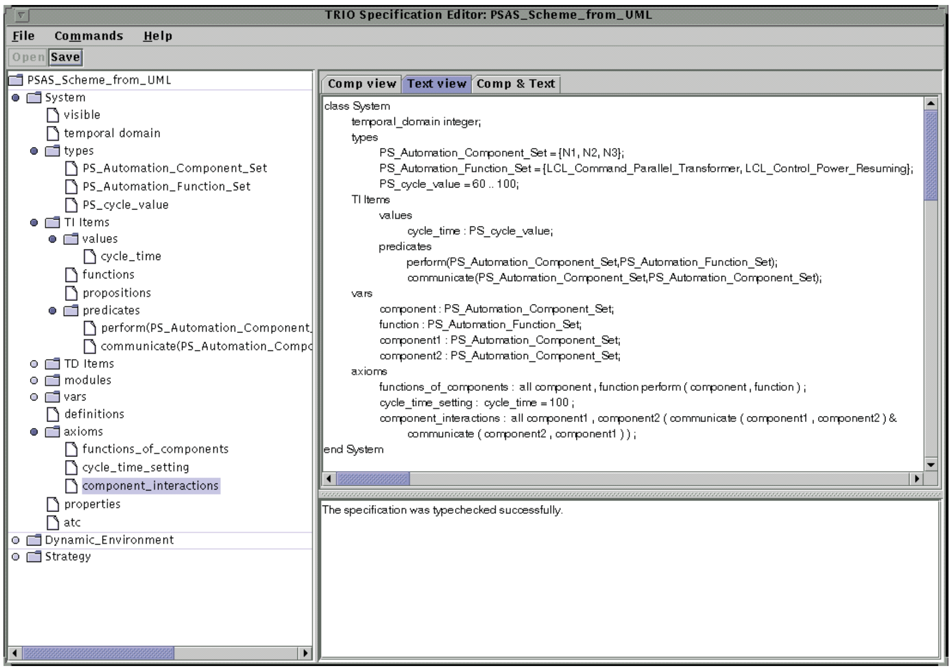
The construction of the classes is described through a set of numbered *actions* that we have defined following the information available in the generic scheme, and that can be applied by the modeller on the specific application using the information available in the instantiated scheme. In the following we present a few examples of actions (again the full set is in [7]), where the numbering respects the original one in [7], for ease of reference, and their application to the PSAS case.

Derivation of the class System. **Sys_Action 1:** in the CD *Structure* the class *Automation System* is composed of a set of *Automation Component* C_i. This set is represented as the domain type *Automation_Component_Set* which is an enumerative range identifying class objects (application of **L1** and **L2**). The type *Automation_Component_Set* is then used to define predicates characterising the class *Automation Component*.
Sys_Action 2 The attribute *cycle_time* of the UML class *Automation System* in the Class Diagram *Structure* is translated into the TRIO Time Independent (TI) value *cycle_time* taking values over the type *cycle_value* (application of **L3**). The range of values of the UML *cycle_time* attribute defines the type *cycle_value* (application of **L4**). Since a single value is also present for the UML attribute, then the axiom *cycle_time_setting* is introduced assigning that value to the item *cycle_time*. Let us now apply the previous actions to the PSAS case, by considering the set of CDs customised over the PSAS application.
Application of Sys_Action 1 According to the customised CD of Fig. 2(A) the PSAS system is composed of three Primary Substation Automation Components. Therefore the domain type *PS_Automation_Component_Set* is introduced which ranges over three values: N1, N2 and N3.
Application of Sys_Action 2 According to the instantiated CD of Fig. 2(A) the PSAS cycle_time is set to 100 ms and the range from 60 to 100 ms. Therefore the domain type *PS_cycle_value* and the TI item *cycle_time* are introduced, as well as the axiom *cycle_time_setting*. Fig. 3 shows the partial specification of the class *System* for the PSAS obtained by applying all the actions.

Derivation of the class DE For what concern the class DE, 9 actions have been defined to specify faults, errors, failures, propagation in the FEF chain and relationship with the affected system elements. As an example we present here only those relative to faults and to their propagation into errors.
DE_Action 1 and its application Enumerative types should be introduced for specifying the possible types of faults, errors, and failure. The application of this action for faults, using the information from the *Fault* model in the PSAS instantiated CDs leads to the enumerative type *PS_Fault_Categories* = { *perm_physical, temp_physical* }.
DE_Action 2, 3 and their application These actions relate FEF elements to system elements, following the information of the *affect* CD association, and require to introduce three TI predicates with associated axioms to formalize the predicate. For faults the predicate is called *fault_affect_component (Fault_Categories, Automation_Component_Set)*, and its application to the PSAS leads to the predicate *fault_affect_component (PS_Fault_-Categories, PS_Automation_Component_Set)*. Assuming *component* is a variable of *PS_Automation_Component_Set* type, the axiom is:

```
fault_model:  all component
        (fault_affect_component(perm_physical, component) &
        fault_affect_component(temp_physical, component));
```

DE_Action 7 The axiom *error_causes* is introduced: it traces back to the cause-effect association in the CD of the FEF chain. An error can be directly caused by a fault, or by the propagation of an error. The axiom states that, at a given instant, an error can affect *function1* performed by component1 if some *t* time units before a fault occurred



```
class System
    temporal_domain integer;
    types
        PS_Automation_Component_Set = {N1, N2, N3};
        PS_Automation_Function_Set = {LCL_Command_Parallel_Transformer, LCL_Control_Power_Resuming};
        PS_cycle_value = 60 .. 100;
    TI Items
        values
            cycle_time : PS_cycle_value;
        predicates
            perform(PS_Automation_Component_Set,PS_Automation_Function_Set);
            communicate(PS_Automation_Component_Set,PS_Automation_Component_Set);
    vars
        component : PS_Automation_Component_Set;
        function : PS_Automation_Function_Set;
        component1 : PS_Automation_Component_Set;
        component2 : PS_Automation_Component_Set;
    axioms
        functions_of_components : all component , function perform ( component , function ) ;
        cycle_time_setting : cycle_time = 100 ;
        component_interactions : all component1 , component2 ( communicate ( component1 , component2 ) &
            communicate ( component2 , component1 ) ) ;
end System
```

```
The specification was typechecked successfully.
```

Fig. 3. Skeleton of the PSAS System class

(a fault of a type that can affect *component1*), or if some *t* time units before an error occurred to a *function2* of *component2*, and *component1* and *component2* communicate (thus allowing error propagation).

```
error_causes:   Alw(all component1, function1
          ( Becomes(error(function1,component1)) ⟶
                  (( perform(component1,function1)   &
                     ex fault_cat, t ( fault_affect_component(fault_cat,component1)   &
                            Dist(Becomes(fault(fault_cat,component1),-t)))   ||
                     ex component2, function2 (communicate(component1,component2)   &
                            perform(component2,function2))   &
                            ex t Dist(Becomes(error(function2, component2),-t))))));
```

Derivation of the class Strategy The set of derivation actions for this class introduces a label for each dependability step in the strategy (*error_recovery*, *error_detection*, and *fault_treatment*), an axiom (*cycle_number*) setting the number of cycles needed to perform the whole strategy, and the property *performance* establishing the duration of the strategy in terms of *cycle_number*.

4.2 Completing the Skeleton with Domain Dependent Knowledge

Once the modeller has derived the TRIO skeleton, the methodology proposes a number of *completion actions*, that provide a set of pre-defined predicates and axioms pertinent to the automation system domain. The modeller will then select the actions that he consider relevant for the application, leading to an enrichment of the partial specification produced in the previous step. Again, only a subset of the actual completion steps are shown here, the full description being in [7].

Sys_Completion1 In a fault tolerant system, any *Automation_Component* C_i may be operational or not (that is to say it is included in the current configuration). The predicate *included(C)* is therefore introduced: it is a TD predicate since a given *Automation_Component* may change its operational status over time.

Sys_Completion2 The axiom *cycle_boundary* is introduced: it formalises which event determines the cyclic evolution of the distributed system. If the cycle is determined by the reception of a periodic_signal, the axiom is naturally expressed by the TRIO operator **NextTime**(F,t), where F is the cycle signal, representing for instance the starting of a new cycle, and t is a term set equal to *cycle_time*:

```
cycle_boundary: Alw(periodic_signal_received ⟷
          ex cycle_t (cycle_t = cycle_time   &   NextTime (periodic_signal_received, cycle_t)) );
```

Sys_Completion3 The (initial, normal, abnormal) behaviour of a distributed automation system is based on message exchange protocols which are formalised by two enumerative types *Received_Messages* and *Sent_Messages* and two TD predicates: *message_received(Received_Messages, Automation_Component_Set)* and *send_message(Sent_Messages, Automation_Component_Set)*.

Sys_Completion4 The axiom label *normal_behavior* is introduced which formalises what the system should do in normal conditions. The actual definition of the axiom will be done at a later stage, when considering application dependent information.

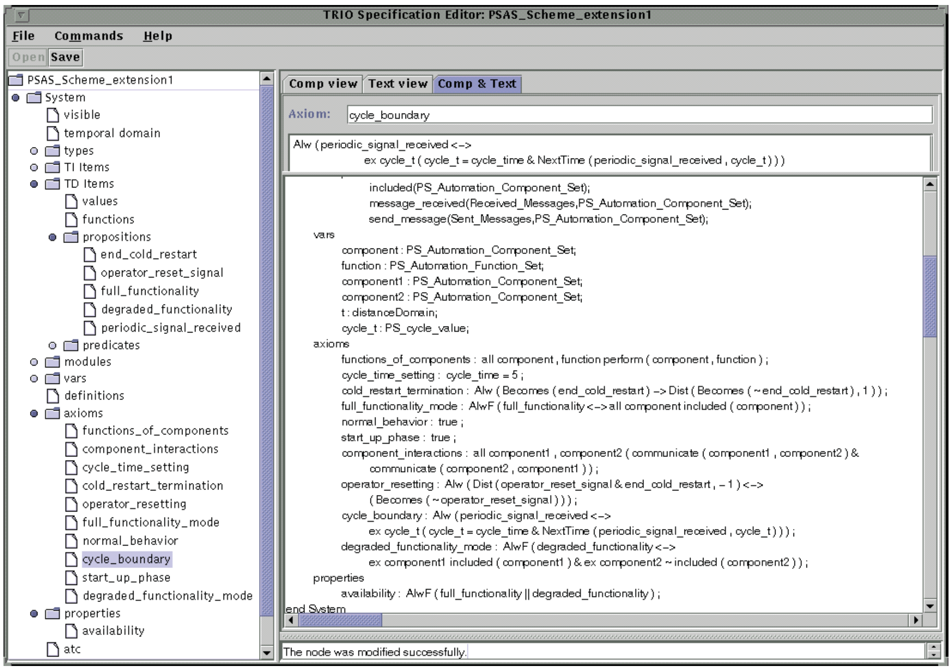
All the completion rules above are considered relevant for the PSAS case and the correspondent axioms and predicates are therefore inserted in the skeleton. This results in the (uncomplete) formalisation of the PSAS *System* class of Fig. 4.

For the completion of the class DE we have chosen to show the definition of temporary fault, that is done in terms of an attribute of faults, called *fault_duration*. A fault is temporary if, given that it occurs at the current time, it will disappear before a time $t1$ smaller than the fault duration parameter, and for all times $t2$ from the current time to $t1$ the fault is active. In TRIO terms:

```
temporary_faults_persistence: Alw(all component (
          Becomes(fault(temp_physical,component)) ⟶
            ( ex t1 ( t1 > 0 &  t1 < fault_duration(temp_physical) &
               Dist(Becomes(∼fault(temp_physical, component)),t1) &
                  all t2 (t2 > 0 & t2 < t1 → Dist(fault(temp_physical, component),t2))))));
```

For the completion of the class *Strategy* we consider the definition of the *error_detection* axiom stating that a component is faulty if there is a potential transient fault in the component, or a permanent fault has been detected.

```
error_detection: Alw ( all component faulty(component) ⟷
          potential_transient_fault(component) | permanent_fault_detected(component));
```



```
cycle_boundary
```

```
Alw ( periodic_signal_received <->
            ex cycle_t ( cycle_t = cycle_time & NextTime ( periodic_signal_received , cycle_t ) ) )
```

```
            included(PS_Automation_Component_Set);
            message_received(Received_Messages,PS_Automation_Component_Set);
            send_message(Sent_Messages,PS_Automation_Component_Set);
    vars
        component : PS_Automation_Component_Set;
        function : PS_Automation_Function_Set;
        component1 : PS_Automation_Component_Set;
        component2 : PS_Automation_Component_Set;
        t : distanceDomain;
        cycle_t : PS_cycle_value;
    axioms
        functions_of_components :  all component , function perform ( component , function ) ;
        cycle_time_setting :  cycle_time = 5 ;
        cold_restart_termination :  Alw ( Becomes ( end_cold_restart ) -> Dist ( Becomes ( ~end_cold_restart ) , 1 ) ) ;
        full_functionality_mode :  AlwF ( full_functionality <-> all component included ( component ) ) ;
        normal_behavior :  true ;
        start_up_phase :  true ;
        component_interactions :  all component1 , component2 ( communicate ( component1 , component2 ) &
            communicate ( component2 , component1 ) ) ;
        operator_resetting :  Alw ( Dist ( operator_reset_signal & end_cold_restart , - 1 ) <->
            ( Becomes ( ~operator_reset_signal ) ) ) ;
        cycle_boundary :  Alw ( periodic_signal_received <->
            ex cycle_t ( cycle_t = cycle_time & NextTime ( periodic_signal_received , cycle_t ) ) ) ;
        degraded_functionality_mode :  AlwF ( degraded_functionality <->
            ex component1 included ( component1 ) & ex component2 ~ included ( component2 ) ) ;
    properties
        availability :  AlwF ( full_functionality || degraded_functionality ) ;
end System
```

```
The node was modified successfully.
```

Fig. 4. Completion of the PSAS system class

4.3 Completing the specification with application dependent knowledge

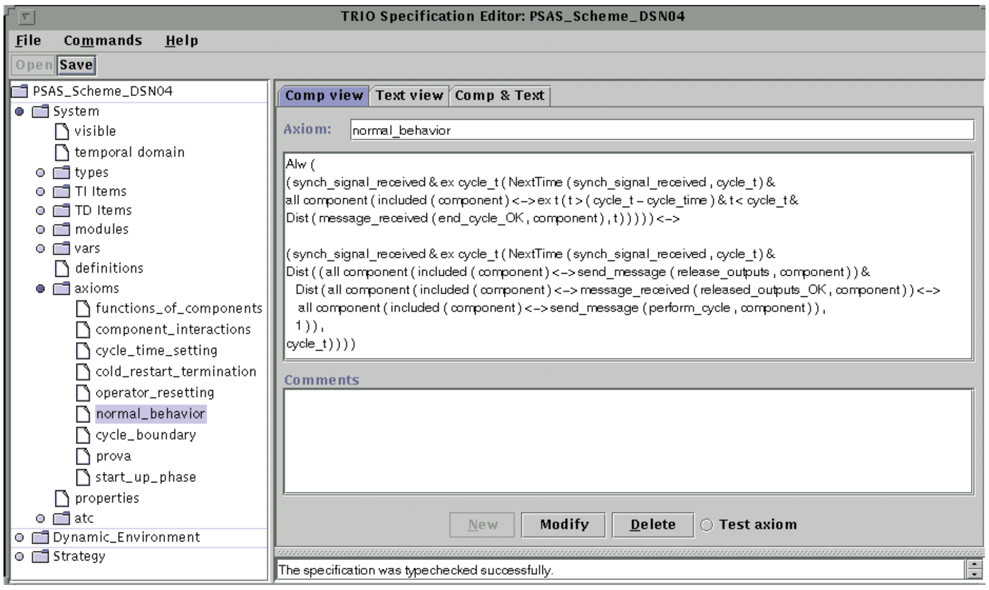
The last phase of the TRIO specification construction includes the full definition of axioms introduced only as labels in the previous phase, so as to obtain a complete (or closed) TRIO formalisation, and possibly the addition of new axioms and properties which are application-specific. Observe that in the first step we have already used application dependent information, but it was information readily available in the instantiated CD scheme (for example the types of faults and components), while in this final step also design level information is needed. As an example consider the following:

(Re)application of Sys_Action2 The application of this action in step one led to the assignment of 100 time units to the TRIO item *cycle_time* (see Fig. 3). Considering that the corresponding attribute of the CD has an assigned value of 100ms, a choice of 100 is indeed correct, but it is definitely not the most convenient one from a computational point of view. In TRIO, as in all temporal logic, it is wise to choose the coarsest possible granularity for time. By considering all the events that involve the item *cycle_time* in the design, a choice of 20ms per time unit has been considered appropriate, resulting in an assignment of 5 time units to the item, through the axiom *cycle_time_setting*.

(Re)application of Completion2 The definition of axiom *cycle_boundary* is modified, based on a proposition *synch_signal_received*, that represents the external synchronisation signal received by a task coordinating the activities of the *PS Components*:

cycle_boundary: **Alw** (synch_signal_received $\longleftrightarrow$
  ex cycle_t (cycle_t = cycle_time & **NextTime** (synch_signal_received, cycle_t)));

(Re)application of Completion4 The normal behaviour of the PSAS is described in terms of a message exchange protocol assuring its correct and consistent evolution. On reception of each synch signal the PSAS component with the master role must receive an *end_cycle_OK* message from each slave component at a time *t* which is within a cycle time (**Dist**(message_ received(end_ cycle_ OK, component),t)). If the *end_cycle_OK message* is received by each component, then the master component performs a *confirm_cycle* procedure. The confirmation of the last elaborated cycle consists of sending the orders of *release_outputs* and *perform_cycle* to all the slave *Automation Components*. As formalised in Fig. 5 the order of starting elaboration on a new cycle is sent if and only if each component confirms the correct emission of its output via *released_outputs_OK* messages within 20 ms (i.e., at **Dist** equal to 1 time unit).



Axiom: normal_behavior

```
Alw (
( synch_signal_received & ex cycle_t ( NextTime ( synch_signal_received , cycle_t ) &
all component ( included ( component ) <-> ex t ( t > ( cycle_t - cycle_time ) & t < cycle_t &
Dist ( message_received ( end_cycle_OK , component ) , t ) ) ) ) ) <->

( synch_signal_received & ex cycle_t ( NextTime ( synch_signal_received , cycle_t ) &
Dist ( ( all component ( included ( component ) <-> send_message ( release_outputs , component ) ) &
  Dist ( all component ( included ( component ) <-> message_received ( released_outputs_OK , component ) ) <->
   all component ( included ( component ) <-> send_message ( perform_cycle , component ) ) ,
  1 ) ) ,
cycle_t ) ) ) )
```

The specification was typechecked successfully.

Fig. 5. Formalization of the normal behavior protocol.

5 How to analyze the TRIO specification

In the previous section we have shown how the logic specification of an application in the automation domain field can be produced re-using the information present in a Class Diagram description of the application, selecting a number of predicates and axioms among a set of predefined ones, and completing the specification with application dependent knowledge made available by the system designers.

Although the construction of a specification is a relevant step in the definition of a dependable automation system, it is also very important to be able to analyze the specification: in TRIO this is realized through model generation and property proof. In this section we show a few examples of model generation. Model generation produces a timed diagram in which the truth values of the selected predicates are plotted against time, and can be considered as an abstract trace of the system execution, concentrating on the predicates of interest, while property proof amount to proving that a property is valid (that is to say, true for any model) for a given temporal window.

In order to perform the analysis of the PSAS behaviour the TRIO models consistent with the specification may be generated automatically by setting up proof sessions in which a subset of axioms, properties and ATC is selected.

For what concerns the class *System* we consider an example on analysis of the normal behaviour of the PSAS (showing its intended functionality in a fault-free setting), specified by the the axiom *normal_behavior* of Fig. 5, and we ask TRIO to generate all models that represents an execution compatible with the axiom *normal_behavior*. This may lead to too many models: to concentrate on the most interesting ones it is necessary to restrict the focus of the analysis, using Abstract Test Cases. ATCs may be both domain dependent and application dependent. For normal behaviour model generation we concentrate on an initial state characterized by all components being operational, at the instant of time in which the synchronization signal is received (ATC 1), we consider only configurations that are stable (ATC 2) and in a scenario in which all messages are received normally (ATC 3).

(ATC 1) *normal_initialisation*: sets the initial truth-values of system primary attributes, including system configuration (predicates included). The PSAS initialisation establishes that: before the evaluation instant no *PS Automation Components* is included and the synchronization signal is false and that at the evaluation instant all the *PS Automation Components* are included and synchronization signal becomes true.

normal_initialisation:
    AlwP(all components ∼included(components) & ∼synch_signal_received) &
    all components included (components) & synch_signal_received ;

(ATC 2) *stable_configuration*: it is used to restrict the analysis to models in which the components of the system, once included, will not be removed:

stable_configuration: **all** components
    (included(components) ⟶ **AlwF** (included(components)));

(ATC 3) *normal_scenario*: it focuses the generation process only on cases in which each component sends the expected messages in due time, and it chooses a specific timing for message reception. An example temporally confined to the first cycle is given by the following ATC in which all the *end_cycle_OK* messages are received at time 4 and all the *released_output_OK* messages at time 6:

normal_scenario : **all** components
  (included(components) ⟷
    (**Dist**(message_received(end_cycle_OK, components), 4) &
    Dist (message_received (released_outputs_OK , components) , 6) &
    all t (t <> 4 ⟷ **Dist** (∼message_received (end_cycle_OK , components) , t)) &
    all t (t <> 6 ⟷ **Dist** (∼ message_received (released_outputs_OK , components) , t)))) ;

The set-up of the model generation for the normal behaviour case is done through the TRIO graphical interface. Three axioms have been selected: *normal_behavior* as expected, and *cycle_time_setting* and *cycle_boundary* that define the notion of cycle, and whose definition is given in Fig. 4. The three ATCs defined above are selected, so that only models compatible with the three axioms and the three ATCs will be generated.

At this point the TRIO tool asks for the temporal window of reference of the proof: obviously the larger the window, the more expensive is the analysis. Since with this proof we want to observe the normal behaviour of the system, and since the whole system behaviour is defined in term of cycles, it is a natural choice to choose a temporal domain size that is a multiple n of *cycle_time* (that is set to a *cycle_value* equal to 5 for the PSAS). For this proof a value of $n = 2$ has been chosen, leading to a temporal window of 10 time units, that allows to check the behaviour of the system upon the reception of two successive synchronization signal. In general the value of n should be the minimal number of cycles needed to check a certain behaviour, for example when checking a complex dependability strategy the value of n could be given by the system requirements (recovery has to terminate in within k cycle), and we can use model generation to check that this requirement is indeed met.

The model generation of TRIO produces then an execution depicted in Fig. 6: the simulation window shows the truth values over the timeline of the time dependent items of the proof that the modeller has selected for visualization. The model that has been generated corresponds to a "normal_behavior" execution in which each included component has sent an *end_cycle_OK* message, the order *released_output* has been sent to each component, the acknowledge has been received in 20 ms (one time unit) and finally the order to perform the next cycle has been sent to all components.

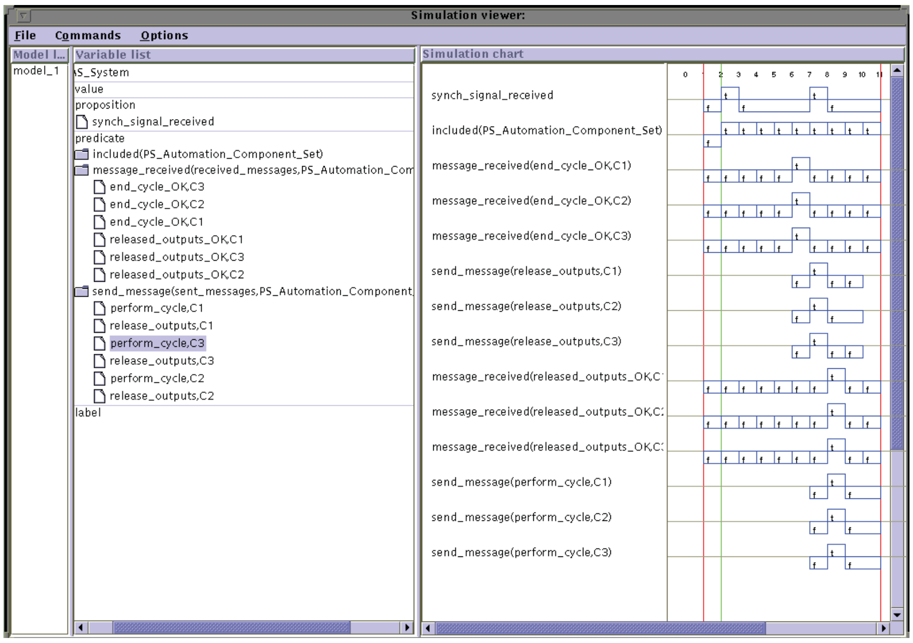

Simulation viewer:
File
Commands
Options
Model l...
model_1
Variable list
AS_System
value
proposition
synch_signal_received
predicate
included(PS_Automation_Component_Set)
message_received(received_messages,PS_Automation_Com
end_cycle_OK,C3
end_cycle_OK,C2
end_cycle_OK,C1
released_outputs_OK,C1
released_outputs_OK,C3
released_outputs_OK,C2
send_message(sent_messages,PS_Automation_Component,
perform_cycle,C1
release_outputs,C1
perform_cycle,C3
release_outputs,C3
perform_cycle,C2
release_outputs,C2
label
Simulation chart
0 1 2 3 4 5 6 7 8 9 10 11
synch_signal_received
included(PS_Automation_Component_Set)
message_received(end_cycle_OK,C1)
message_received(end_cycle_OK,C2)
message_received(end_cycle_OK,C3)
send_message(release_outputs,C1)
send_message(release_outputs,C2)
send_message(release_outputs,C3)
message_received(released_outputs_OK,C
message_received(released_outputs_OK,C
message_received(released_outputs_OK,C
send_message(perform_cycle,C1)
send_message(perform_cycle,C2)
send_message(perform_cycle,C3)
t
f

Fig. 6. The generated model for the normal_behavior proof

The TRIO formalisation of the class *Dynamic Environment* allows to study the effect of faults on a system in which a dependability strategy has been deployed, while the analysis of this class together with the class *Strategy* allows to study the effectiveness of the dependability strategy in limiting the damage due to faults.

As an example we consider the case in which PSAS faults are considered (axiom *fault_model*), there is a full communication structure among components (axiom *component_interaction* defined in Fig. 4) each component can perform any function (axiom *functions_of_components* defined in Fig. 4), and the relationship between faults and errors is set according to axiom *error_causes* (defined in the previous section as a result of **DE_Action7**). Since we want to study the effect of faults we concentrate the focus of this first proof on executions that experience a single fault. The model produced (whose window is not shown here for space reasons) depicts a behaviour in which a single fault propagates to all components, and therefore to all functionalities so that the system is not able to deliver the expected service.

The analysis of this class allows therefore to make explicit the global effect of a chain of local actions (as expressed by the *communicate*, *perform*, and *cause_effect* associations that were already present on the UML CDs and that have been translated into TRIO predicates and axioms) under different fault occurence settings.

6 Conclusions

In this paper we have presented, with the help of a case study, a support to the specification and analysis of dependable automation systems which makes use of UML class diagrams and of the declarative temporal logic TRIO. In the context of formal analysis tools the peculiarity of TRIO lays on the possibility of analysing temporal scenarios underlying the specification in a uniform framework which makes use of the same language for both specifying and querying the system. The TRIO tool may be classified as a temporal theorem prover, like PVS is for higher order logics.

The combined use of UML with formal methods in the functional specification and analysis of software systems has received a great attention by the research community, with the goal of giving a formal semantics to the UML diagrams, usually through translation into another formal language (there is a very large body of literature on the topic, see for example the work of the Precise UML group [23]).

In this paper we do not propose a translation, but a pre-defined set of temporal logic specifications that have been associated to a pre-defined description of an automation system through a set of UML Class Diagrams. The proposed approach is meant to provide requirement reuse, a topic that, following the work on patterns [8] for design reuse, is gaining increasing interest: in [14] UML based patterns (mainly CD and Statecharts) are defined for embedded system requirements, and the work is extended in [15] to include properties specified in the linear temporal login LTL of SPIN [13].

The novelty of our contribution is in identifying a three-steps approach in which the costruction of the formal specification follows partially a derivative style, and partially a selective style. It is assumed that the analysist still needs to play an important decision role in the analysis, whilst the tool provides him with a methodological support.

The specification support provided here is three steps: the TRIO class structure and a number of initial TRIO items and types are (manually) derived from a UML CD description of the system; this partial specification is then augmented in a second step with a number of domain dependent information; while in the third step the specification is completed using application dependent knowledge. The role of the modeller increases in the three steps: in the first one he only has to apply the predefined actions by extracting information from the instantiated CD diagrams, in the second step he will have to select the subset of predicates and axioms that are considered relevant for the application, while in the third step he has to apply his expertise to define all axioms and additional predicates needed to complete the specification.

Writing TRIO formulae requires indeed a certain skill. To make the use of TRIO transparent to the user the work in [17] proposes the automatic generation of TRIO formulae from annotated UML Statecharts (in the context of real-time systems): this result could be integrated in our approach, especially when the modeller is fluent in UML Statecharts, so that certain parts of the specification can be automatically produced.

The paper also provides a (limited) support to the formal analysis, an activity which requires skill not only to define the system, but also to drive the proof sessions to avoid an explosion of complexity. The methodological lines presented in the paper represents a preliminary result: a support to the identification and definition of ATC, and to the definition of the appropriate temporal window for the analysis is an interesting topic for future research. In particular it is still to be investigated to which extent the "guided approach to specification" described in this paper can be coupled with a "guided approach to analysis". In the paper we have presented examples of analysis: analysis of a logic specification is an incremental activity, and the space limitations allows only the exemplification of limited steps of the analysis activity.

The methodological approach has been exemplified on a case study taken from control systems of electric distribution network. However, it seems reasonable to consider the proposed three steps methods applicable also to other applications in the automation system domain, given the generality of the closed loop execution model considered.

Finally, the TRIO specification has been built starting from an ad-hoc description of the dependability aspects of an automation systems. Following the work on the UML profiler for Performance and Schedulability [20] it is likely that an extension to include dependability aspects will be made available in the near future [22]: it will then be necessary to adapt the proposed CD scheme to the new standard.

References

1. S. Bernardi and S. Donatelli. Building Petri net scenarios for dependable automation systems. In *Proc. of the* 10th *International Workshop on Petri Nets and Performance Models*, pages 72–81, Urbana-Champain, Illinois (USA), September 2003. IEEE CS.
2. S. Bernardi, S. Donatelli, and G. Dondossola. Methodology for the generation of the modeling scenarios starting from the requisite specifications and its application to the collected requirements. Technical report. Deliverable D1.3b - DepAuDE Project 25434, June 2002.
3. A. Bertani, E. Ciapessoni, and G. Dondossola. Modular TRIO Manual and Guidelines, Tutorial Package. Part I-II, Deliverable D3.4.1 of the FAST Project No. 25581, May 2000.

4. International Electrotechnical Commission. IEC-60300-3-1: Dependability Management. IEC, 3 rue de Varembé CH 1211 Geneva, Switzerland, 2001.
5. G. Deconinck, V. De Florio, R. Belmans, G. Dondossola, and J. Szanto. Integrating recovery strategies into a Primary Substation Automation System. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)*, pages 80–85, San Francisco, California (USA), June 2003. IEEE Computer Society ed.
6. DepAuDE. EEC-IST project 2000-25434. http://www.depaude.org.
7. G. Dondossola. Dependability requirements in the development of wide-scale distributed automation systems: a methodological guidance. Technical report. Deliverable D1.4 - DepAuDE IST Project 25434, February 2003.
8. Gamma E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
9. The FAST toolkit homepage. http://www.prover.com/fast.
10. M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Trans. Softw. Eng. Methodol.*, 3(4):308–339, October 1994.
11. Dondossola G. and Botti O. System fault tolerance specification: Proposal of a method combining semi-formal and formal approaches. In *Fundamental Approaches to Software Engineering, FASE 2000*, volume 1783, pages 82–96. Springer, January 2000.
12. C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.
13. J. Gerard Holzmann. *SPIN Model Checker: the Primer and Reference Manual*. Addison Wesley Professional, 2004.
14. S. Konrad and B.H.C. Cheng. Requirements Patterns for Embedded Systems. In *In Proc. of the Joint International Conference on Requirements Engineering (RE02)*, Essen, Germany, September 2002. IEEE CS.
15. Sascha Konrad, Laura A. Campbell, and Betty H. C. Cheng. Adding formal specifications to requirements patterns. In C. Heitmeyer and N. Mead, editors, *Proceedings of the IEEE Requirements for High Assurance Systems (RHAS02)*, Essen, Germany, September 2002.
16. J. C. Laprie. Dependability – Its attributes, impairments and means. In B. Randell, J.C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 3–24. Springer Verlag, 1995.
17. L. Lavazza, G. Quaroni, and M. Venturelli. Combining UML and formal notations for modelling real-time systems. In *Proc. of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT Int. symposium on Foundations of software engineering*, pages 196–206, Vienna, Austria, 2001. ACM Press.
18. D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.*, 13(4):365–398, November 1995.
19. M.K. Molloy. Performance analysis using Stochastic Petri Nets. *IEEE Transaction on Computers*, 31(9):913–917, September 1982.
20. OMG. UML Profile for Schedulability, Performance, and Time Specification. http://www.omg.org, March 2002.
21. OMG. UML Specification: version 1.5. http://www.omg.org, March 2003.
22. A. Pataricza. From the General Ressource Model to a General Fault Modeling Paradigm ? In J. Jürjens, M.V. Cengarle, E.B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 163–170. Technische Universität München, Institut für Informatik, 2002.
23. The Precise UML Group. http://www.puml.org.
24. The European MAFTIA Project. Web page: http://www.research.ec.org/maftia.