FROM FAULT TREE ANALYSIS TO MODEL CHECKING OF LOGIC CONTROLLERS

Israel BARRAGAN SANTIAGO [(1)] and Jean-Marc FAURE [(1), (2)]

[(1)] LURPA – ENS Cachan – 61, Avenue du Président Wilson, 94230 Cachan, France
{barragan, faure}@lurpa.ens-cachan.fr
[(2)] Institut Supérieur de Mécanique de Paris (SUPMECA) – 3 rue Fernand Hainaut, 93407 Saint-Ouen, France

Abstract: This paper proposes a method enabling to state formal properties of a logic controller, a prerequisite for formal verification using model-checking, from a fault-tree analysis taking into account both the controlled process and the controller. Invariants, untimed and timed properties are considered and illustrated thanks to an example. The aim of this method is to ease formal properties design and to bridge the gap between fault forecasting and fault removal for automated systems. *Copyright © 2005 IFAC*

Keywords: Dependability, Fault forecasting, Fault removal, Fault tree analysis, Formal verification, Temporal logic.

1. INTRODUCTION

Dependability of automated systems is a very important concern in numerous industrial fields. Classical dependability attributes are reliability, safety and availability, and can be improved thanks to lots of methods ranked usually in four categories: fault prevention methods, fault removal methods, fault tolerance methods, and fault forecasting methods (Laprie, 1992). This paper focuses only on fault forecasting and on fault removal.

Ensuring dependability of automated systems requires to take into account not only the random faults produced by failures of physical components of the process (process dependability) but also the faults issued from control algorithms (controller dependability). These last ones come from designer's errors or misinterpretation of the control requirements and behave as systematic faults. Only controller faults will be addressed in this paper.

An efficient way to avoid controller faults consists in using formal verification techniques during design and implementation of control algorithms (Faure and Lesage, 2001). These fault removal methods are aimed at checking whether a given controller satisfies (or does not satisfy) the properties required for the control of the process.

The most usual formal verification technique is named model-checking and relies on state automata theory (Bérard, *et al.*, 2001). Its principle is to check whether a formal property holds on a state model of the system. Hence, formal verification of controller using model-checking implies to build a state model of the controller as well as to write the formal properties that express in a formal way the application requirements that must be satisfied by the controller. The formalism used to write the controller properties depends upon the selected model-checking tool; choosing the untimed model-checker NuSMV (Cimatti, *et al.*, 2000), for instance, will lead to employ the CTL (Computation Tree Logic) temporal logic, while properties proof thanks to the timed model-checker UPPAAL (Uppaal homepage) will enable to represent formal properties in the form of timed automata.

[*] Israel Barragán is financed by the Mexican Council of Technology CONACYT.

Unfortunately properties formalization is a difficult task because the application requirements are expressed in industry in a quite informal way, i.e. some sentences in natural language or drawings, but never with sound mathematical statements. Moreover the formalisms used by the model-checking tools (temporal logic or timed automata) are totally unknown by automation engineers.

The objective of the method presented in this paper is to overcome this problem by facilitating the elaboration of formal properties. Other works like (Filkorn, 1999) have been developed with the same purpose but although those methods may usefully help designers looking for formal properties, they require some experience and are not directly connected to any existing industrial method for dependability improvement. Conversely the work presented in this paper proposes a methodology enabling to obtain formal properties for model-checking of control algorithms by means of an analysis technique commonly employed in industry for critical systems design: fault-tree analysis.

This work enables therefore to bridge the gap between a widely used fault forecasting method: fault tree analysis, and a fault removal method: model checking. Our aim is to take benefit of the results of a fault tree analysis, developed from the application requirements, to elaborate formal properties. Using fault tree analysis as the starting point of the method will facilitate its acceptance by industrial users. This paper is structured as follows. Section 2 sketches the bases of Fault Tree Analysis (FTA) and presents some recent developments of this fault forecasting technique. The proposed method is explained in section 3 and is exemplified in Section 4 thanks to a simple mechatronics system. Conclusions and prospects are discussed in the last section.

2. FAULT TREE ANALYSIS

Since its development in 1960 by the American company Bell Telephone, a lot of technical and scientific works have been reported in the literature about fault tree analysis. Today it is a well known technique, widely used in companies for critical systems design. This section addresses certain fault trees concepts and discusses some additional variants of the traditional fault tree method that improve its application scope.

2.1 Fault Tree fundamentals

Fault Tree Analysis aims to find out all the associated sets of basic events (cut sets) in the system that could cause that a given top event (a system failure of some kind) occurs. Minimal Cut Sets are the smallest combinations of basic events being able to lead to the undesirable event. The events are termed "faults" if they are initiated by other events and are termed "failures" if they are the basic initiating events (US N.R. Commission, 1981). A basic event is an event that is not developed further in the analysis. The connections between the various identified basic events are carried out by means of logical gates. The two basic gate categories are the AND-gate and the OR-gate. Nowadays, it is possible to find a large offer of software tools to generate automatically fault trees, to find minimal cut sets and to calculate failure probabilities (Brooke, 2003). As our objective is to find out systematic faults of controller, failure probabilities computation will not be considered in what follows. Focus will be put only on logical relationships between events.

2.2 Dynamic Fault Trees

Classical fault trees are clearly structured, but unable to model some aspects of real systems behaviour, such as event ordering information. A Fault Tree is called dynamic if it integrates relative time and sequences of events, which force a particular event to happen before or after another. Several research works have been carried out to enhance dynamic fault trees capacities. Thus (Bozzano and Villafiorita, 2003) proposes a method to generate automatically fault trees with minimal cut sets containing event ordering information. (Cepin and Mavko, 2002) uses the house events to follow the evolution of a fault tree with respect to time. (Dugan, 1999) proposes the extension of fault trees with additional gates, which can model functional dependencies or primary-spare relationships. The works mentioned previously focus on faults of the process whereas our work deals with faults of the controller. It is the reason why we prefer to use the dynamic gates Exclusive OR with condition and Priority AND (US N.R. Commission, 1981) depicted in figure 1.

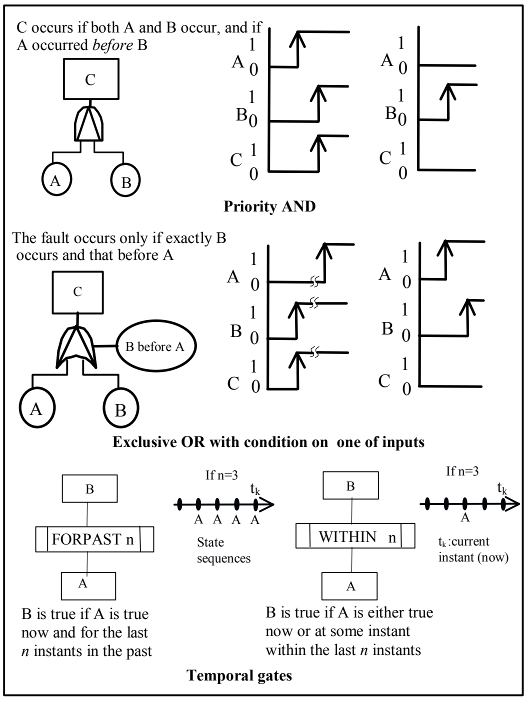
C occurs if both A and B occur, and if A occurred *before* B
C
A
B
A
B
C
1
0
Priority AND
The fault occurs only if exactly B occurs and that before A
B before A
Exclusive OR with condition on one of inputs
If n=3
t_k
A A A A
State sequences
B
FORPAST n
A
WITHIN n
t_k :current instant (now)
B is true if A is true now and for the last *n* instants in the past
B is true if A is either true now or at some instant within the last *n* instants
Temporal gates

Fig. 1. Dynamic and temporal gates

2.3 *Temporal Fault Trees*

While dynamic gates focus on relative time and sequential behaviour, temporal gates specify actual time intervals allowing to state physical delays between events. The work developed by (Palshikar, 2003) proposes the creation of other special gates to describe temporal systems. The term Temporal Fault Trees (TFT) is coined for this model. TFT notation allows the user to easily specify the temporal dependence between events and preserves the simple, qualitative and visual nature of the fault trees. The semantics of this series of additional temporal gates is defined in terms of the past-oriented linear propositional temporal logic (PLTLP). The PLTLP includes the usual non-temporal operators: ¬ (not), ∨(or), ∧(and), → (implies), and ↔ (if and only if), as well as several instance-oriented past temporal operators: O_n^-(PREVn), $\Box_n^-$(FORPASTn), $\Diamond_n^-$(WITHINn), U_n^-(UNTIL-PAST), where n is a positive integer. One unary temporal gate is proposed for each one of these temporal connectives. Figure 1 depicts only the FORPASTn and WITHINn gates that will be employed in the example of section 4. More details on temporal fault trees are available in the previously mentioned reference.

3. FORMAL PROPERTIES ELABORATION

3.1 *Method overview*

The basic idea of this method, depicted in figure 2, is to obtain formal properties required to perform a fault removal method: model-checking of a controller, from the results of a fault forecasting method: fault tree analysis. More precisely, this method includes two steps:

1) Design of the fault tree

This fault tree is aimed at describing all the causes of an undesirable event. In the case of automated systems, some of these causes are physical components failures; other ones are issued from the controller. The resulting fault tree includes static gates, dynamic gates, if event ordering description is necessary to describe a fault, and temporal gates if timing constraints must be taken into account. In order to elaborate formal properties related only to controller variables and not to physical components state, any controller fault must be combined with physical failures using only OR logical gates. This enables to decouple these two kinds of faults.

2) Formal properties elaboration

Once the controller faults have been extracted from the fault tree obtained at the previous step, it is possible for each of them to give a formal property stating how the controller must behave so as not to generate this fault.

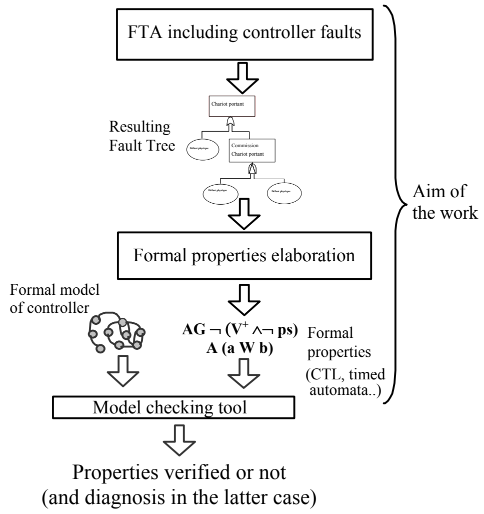
FTA including controller faults
Chariot portant
Resulting Fault Tree
Défaut physique
Commission Chariot portant
Défaut physique
Défaut physique
Aim of the work
Formal properties elaboration
Formal model of controller
AG ¬ (V⁺ ∧¬ ps)
A (a W b)
Formal properties (CTL, timed automata..)
Model checking tool
Properties verified or not (and diagnosis in the latter case)

Fig. 2. Objective of the method

It matters to highlight indeed that the general form of a property to check is in an informal fashion *"The controller never generates the considered fault"*. Formal properties will be written in CTL (Computation Tree Logic) temporal logic for static and dynamic gates or in the form of timed automata if temporal gates are used to describe the fault, for these two formalisms are commonly used by untimed and timed model-checkers. This second step is detailed hereafter.

3.2 Obtaining formal properties from fault tree gates

As previously mentioned, model-checking uses a state automaton model of the controller to check properties. This is the reason why formal properties will refer to *states* and *paths* of this automaton by using *state quantifiers* **F** and **G** and *path quantifiers* **A** and **E**:

- **F** φ means that the property φ holds for some state within a path
- **G** φ means that the property φ holds for all the states within a path
- **A** φ means that the property φ holds for all the paths starting from the current state
- **E** φ means that the property φ holds in some path starting from the current state.

The general forms of formal properties that can be deduced from fault tree gates are given below.

Static gates. The output fault of a static gate is merely a combinatory expression of its inputs. It doesn't depend on time or on events sequence. The controller property to verify must therefore state that for all the paths and for all the states of the state automaton modelling the controller this combinatory expression is never true, i.e. in a formal fashion:

AG ¬[Combinatory expression issued from the gate].

This property is an invariant that requires to consider only present values of controller variables. (Henry and Faure, 2003) illustrates the use of this property for checking invariant safety properties in PLC (Programmable Logic Controller) programs.

Dynamic gates. Two cases will be considered: priority AND gate and exclusive OR gate with condition.

According to the definition of the priority AND gate, the fault happens if its input basic events occur in a specific order. In the figure 1, the output fault happens when the basic event "a" occurs before "b". The formal property deduced from this dynamic gate must therefore state that this sequence of events must never occur within the state automaton modelling the controller, i.e. in a formal way:

$$\mathbf{AG} \neg (a \Rightarrow \mathbf{EF}\ ab).$$

This formal statement means that there is no state where "a" is true and from which starts a path within which stands an other state where "ab" is true[1]. This property is an invariant that requires to consider both present and future values of controller variables.

In the case of an exclusive OR gate with condition, the fault happens if the conditioned input is true while the other input is false. The formal property that can be deduced from this gate states that this proposition is never true within the state automaton modelling the controller, i.e. in a formal way:

$$\mathbf{A}\ (\neg b\ \mathbf{W}\ a)$$

where **W** represents the weak until operator (φ **W** Ψ holds if and only if φ holds as long as Ψ does not hold). In the studied case, that means that for all the paths, the variable "b" is never true ($\neg$b) until the variable "a" becomes true. Figure 3 summarizes the previous results.

Temporal gates. The semantic of temporal gates in (Palshikar, 2003) is defined in terms of PLTLP but this semantic is not recognized by usual timed model checkers like Uppaal. Therefore, properties issued from temporal gates will be depicted as timed automata, which is a format accepted by timed model checkers. This will be illustrated in the next section.

4. EXAMPLE

The proposed method will be illustrated by means of an example: control of the test station. This station is the second one of a Bosch mechatronics system whose aim is to assembly/disassembly gear wheels. The main function of this station is to test whether a plain bearing is housed into the bore of the gear wheel. Only automatic operations will be considered in the following.

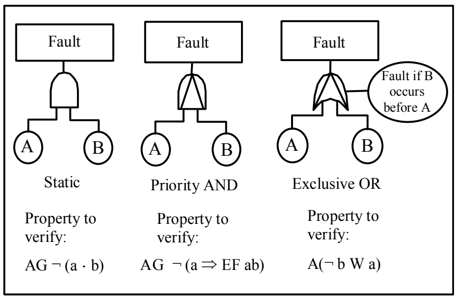

Fig. 3. Properties to verify derived from some gates

4.1 The process

At the initial position (receiving position at the left side of the station) the carriage waits for a part from the previous station. The transfer signal issued by this station starts the process. Then the carriage moves right to a testing position where the presence or the absence of a plain bearing is detected. It passes under a detection sensor on this way; this sensor aims at detecting the absence or the presence of a part on the carriage.

If there is no part on the carriage, it runs back to the initial position and the station demands a new gear wheel. In the opposite case, the process goes on and the carriage stops at the testing position. Then a pneumatic cylinder scans the bore in the gear wheel; the signal delivered by the cylinder switch indicates whether a plain bearing is in the bore or not. Then the carriage moves to the transfer position where the gear is transferred to the following station by means of a rotary/lift gripper.

The carriage returns back to the receiving position and the process starts again. A PLC controls the global process. Its inputs and outputs are given in figure 4.

The following sections present the design of fault trees corresponding to two different undesirable events as well as controller properties derived from these fault trees.

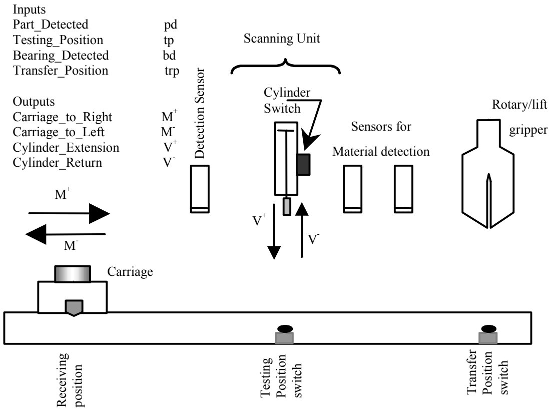

Fig. 4. Diagram of the test station

[1] This property is written assuming that a is a persistent fault, i.e. that remains always true once occurred. Taking into account transient faults implies to rewrite the property as follows: a $\Rightarrow$ EF b. Therefore two definitions of the priority AND gate (a followed by a.b or a followed by b) will lead to two different formal properties. Only the first one, closer to the usual definition of the AND gate, will be kept in the rest of this paper.

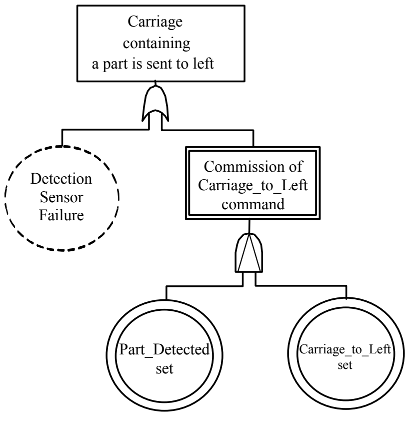
Carriage containing a part is sent to left
Detection Sensor Failure
Commission of Carriage_to_Left command
Part_Detected set
Carriage_to_Left set

Fig. 5. Fault tree for “Carriage containing a part is sent to left”

4.2 First case: fault in carriage control

In this analysis, we consider events related to physical components or to controller or that are combinations of these two kinds of events (edged respectively in fault trees by a dot line, a double line, and a simple line).

The undesirable event to analyse is “carriage containing a part is sent to left” (to the initial position). The tree depicting this fault, shown in figure 5, exhibits two primary faults linked by an OR logical gate. The carriage with a part is sent to the left because the detection sensor has failed or because the controller failed in the signal interpretation and sends the carriage back.

“Commission of Carriage_to_left command” is the result of a Priority AND gate. This fault occurs if at first the signal of the detection sensor is set, then the controller sends back the carriage. Note that we include the normal event “Part_Detected set”. This is essential here to derive the formal property that can model effectively the event of an erroneous commission of the controller output. Based on the results of section 3.2, the controller property to check is:

$$\mathbf{AG}\ \neg\ (\ pd \Rightarrow \mathbf{EF}\ (pd\ .\ M^{-})) \qquad (1)$$

4.3 Second case: fault in testing cylinder control

The second undesirable top-level event to analyze is “Testing cylinder failure”. The corresponding fault tree is depicted in figure 6. Two primary causes lead to the top-level event. The first one “Commission of Cylinder_Extension with Testing_Position false” is a controller fault that means that the controller sets the cylinder extension when the carriage with a part is in any other position than the testing position.

The other fault means that there is no cylinder extension. This may come from a physical (mechanical or pneumatic) failure or because the controller fails to set the cylinder extension output in a time interval of n units starting from the event "Testing_Position is set”. Hence we use the temporal gates FORPASTn and WITHINn in the fault tree.

Elaboration of formal properties. The root of this fault-tree is the output of an OR logical gate. One input of this gate is itself output of an OR gate. This structure enables to state that the undesirable event will never occur if no physical failure occurs and if none of the following controller faults occurs:

- Commission of Cylinder_Extension with Testing_Position false (F1)
- Omission of the Cylinder_Extension command (F2)

F1 is the output of an AND gate. The property to verify, derived from this gate, can be formally stated: **AG** ¬(Commission of Cylinder_extension with Testing_Position false). Replacing textual expressions by the input and output variables of the control program leads to:

$$\mathbf{AG}\ \neg\ (V^{+} \wedge \neg tp) \qquad (2)$$

We propose two ways to obtain a formal property from the fault F2. The first one is mandatory when untimed model-checking is employed and consists in treating this fault as a dynamic one and not as a temporal one, i.e. taking into account only the relative order of events and not the physical delay between events. Given this reinterpretation, the fault F2 occurs if Cylinder_extension is not set after Testing_Position is set. Consequently the property to check can be expressed as follows: “when Testing_Position is set then the controller must set Cylinder_extension”, that leads to the following formal CTL statement:

$$\mathbf{AG}\ (tp \Rightarrow \mathbf{AF}\ V^{+}) \qquad (3)$$

meaning that for each state where tp is true all the paths starting from this state will contain a state where V^{+} is true.

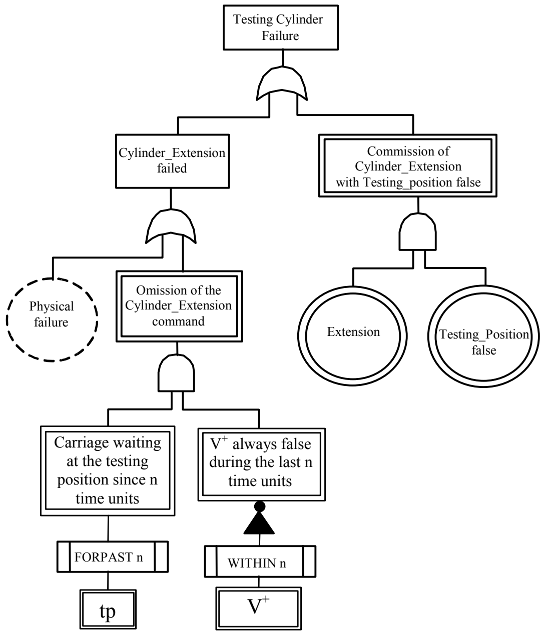
Testing Cylinder Failure
Cylinder_Extension failed
Commission of Cylinder_Extension with Testing_position false
Physical failure
Omission of the Cylinder_Extension command
Extension
Testing_Position false
Carriage waiting at the testing position since n time units
V+ always false during the last n time units
FORPAST n
WITHIN n
tp
V+

Fig. 6. Fault tree for “Testing cylinder failure”

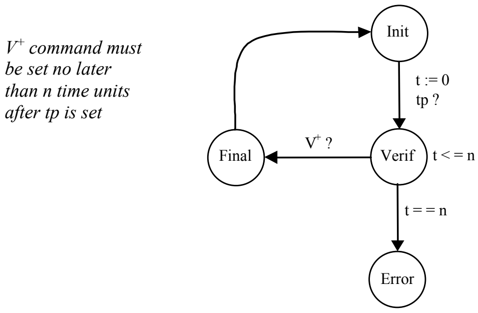
V^+ command must be set no later than n time units after tp is set
Init
t := 0
tp ?
V^+ ?
Final
Verif
t < = n
t = = n
Error

Fig. 7. Timed automaton for the temporal property

The second solution is to express the property in the form of a timed automaton. Timed automata is a modeling language that allows to describe the behavior of systems by means of finite state automata, extended with clocks and time constraints (Alur and Dill, 1994). Timed properties expressed in this way can be verified with a timed model checker like Uppaal (Zoubek *et al.*, 2003).

The corresponding automaton for the studied property is shown in figure 7. The clock of the system is the variable *t* and is increased at the same rate as time. From the initial state of this automaton, only one transition leads to an intermediate state *Verif* if Testing_Position (tp) is true. In this transition the clock is reset. From this intermediate state, where the value of the clock must be smaller than or equal to n, two transitions are possible, the first one arriving to *Final* if V+ is true, (the controller set Cylinder_Extension within the n units of time), the second one to *Error* if *t=n* (the controller failed to set Cylinder_Extension within these n units). The property to check holds if this *Error* state is never reached. The formalization of the property in the form of a state automaton keeps better the semantics of temporal fault trees and must be prioritized. The first solution is an useful alternative when only untimed model-checking is used.

5. CONCLUSIONS

Model checking is a quite popular technique for formal verification of controllers and an useful tool when designing dependable automated systems. However its application requires that the properties to verify are written in a formal way. This paper proposes to facilitate this task by means of a preliminary fault-tree analysis taking into account not only random faults coming from physical components but also systematic controller faults. Dynamic and temporal gates enable to model complex cause-consequence relationships taking into account event ordering information, and real-time constraints. The simple case-study presented illustrates the interest of this method.

Several prospects can be drawn from this work. First of all, it matters to determine the minimum set of gates needed to describe all potential controller faults; only some usual or promising gates have been dealt with indeed. Hence several case studies must be performed to determine whether this set is sufficient or not. These studies may lead to define new gates able to describe more complex events ordering information and time constraints than the ones presented. Obtaining minimal cut-sets of fault-trees including dynamic and temporal gates is also a challenging issue that deserves to be addressed. At last, robustness and scalability of the properties elaboration method will have to be evaluated thanks to larger examples.

REFERENCES

Alur, R. and D.L. Dill (1994). A Theory of Timed Automata. *Theoretical Computer Science*, **Vol. 126**, pp. 183-235.

Bérard, B. et al (2001). Systems and Software Verification. *Model-Checking Techniques and tools.* Springer.

Brooke, P. and R. Paige (2003). Fault trees for security system design and analysis. *Computers & Security*, **Vol 22**, n° 3, pp. 256-264.

Bozzano, M. and A. Villafiorita (2003). Integrating Fault Tree Analysis with Event Ordering Information. *In Proceedings of ESREL 2003*, pp. 247-254, June 15-18, Maastricht, The Netherlands.

Cepin, M. and B. Mavko (2002). A dynamic fault tree. *Reliability Engineering and System Safety*, **N° 75**, pp. 83-91.

Cimatti, A., E. Clarke and M. Roveri (2000). NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4), pp. 410–425.

Dugan, J.B. and K.J. Sullivan (1999). Developing a low-cost, high-quality software tool for dynamic fault tree analysis. *Transactions on Reliability*, pp. 49-59.

Faure, J.M. and J.J. Lesage (2001). Methods for safe control systems design and implementation. INCOM'2001, CDRom paper, 6 pages, September 20-22, Vienna, Austria.

Filkorn, T. (1999). Formal verification of PLC-programs. *Proceedings of the 14th IFAC*, July 5-9, Beijing, P.R. China.

Henry, S. and J.M. Faure (2003). Elaboration of invariant safety properties from fault-tree analysis. *Proceedings of IMACS-IEEE "CESA'03*, 6 pages, July 9-12, Lille, France.

Laprie, J.C. (1992). *Dependability: basic concepts & terminology*, Springer-Verlag Editions.

Palshikar, G.K. (2003). Temporal Fault Trees. *Information and Software Technology*, **n° 44**, p137-150.

Zoubek, B., J.M. Roussel and M. Kwiatkowska (2003). Towards automatic verification of ladder logic. *Proceedings of IMACS-IEEE "CESA'03*, 6 pages, July 9-12, Lille, France.

UPPAAL model checker homepage. http://www.uppaal.com.

US Nuclear Regulatory Commission (1981). Fault Tree Handbook. *Technical Report NUREG-0492, Washington, DC.*