A General Framework for Applying Safety Analysis to Safety Critical Real-Time Applications Using Fault Trees

Vasilis C. Gerogiannis[1,2] Ioannis E. Caragiannis[1,3] Manthos A. Tsoukarellas[1,4]

[1]Advanced Informatics Ltd., 35 Gounari Ave., 26221 Patras, Greece
[2]Department of Mathematics, Sector of Informatics, University of Patras, Greece
[3]Computer Engineering and Informatics Department, University of Patras, Greece
[4]Technological Edutational Institute of Patras, Greece

e-mail: {V.Gerogiannis | caragian | manthos}@advinfo.pat.forthnet.gr

Abstract

This paper presents a general framework for analyzing the safety aspects of complex safety critical real-time applications. The proposed framework is based on the well-established Fault Tree Analysis (FTA) technique and provides a systematic way for handling fault trees, as well as, discovering any hazardous conditions that may arise. It also provides the background for the development of automated software safety analysis tools oriented to a broad set of programming languages or specification/ design notations for which fault tree templates are (or will be) available. Such a tool can be used to assess software safety at most phases of software life cycle. The paper presents SAFELAND tool which follows framework disciplines and performs code-based fault tree analysis on safety critical real-time applications written in several idioms of the C programming language.

1. Introduction

Safety critical systems (e.g., in avionics and nuclear industries area) are characterized by the fact that consequences of possible failures are serious and may involve grave danger to human life or property. Thus, in many cases, safety critical systems become more crucial than conventional real-time systems. Many industries have a long tradition over applying techniques to analyze the behavior of their software products regarding the presence of failures (safety analysis) [10, 12], as well as, their behavior with respect to strict timing constraints (schedulability analysis) [9].

This paper presents an open and general framework for applying software safety analysis: as far as automated schedulability analysis tools are concerned, more interested readers may refer to [6, 8]. The framework theoretical basis is the well-established Fault Tree Analysis (FTA) approach [3, 12, 13, 19]. The framework structure consists of distinct phases which can be applied to assess software safety at most stages of the software development life cycle, from the requirements specification up to the implementation.

The framework provides the necessary background in order to develop automated safety analysis tools towards a broad set of programming languages or specification/design notations for which fault tree templates are (or will be) available [13]. Such a tool is SAFELAND, which aims to analyze applications written in ANSI C programming language. A subset of the SAFELAND algorithms has been developed in the context of the ESPRIT project 20899-OMI/ANTI-CRASH[1] [4, 5].

The rest of the paper is structured as follows. In section 2, the issues that have led to the choice of FTA, as the backbone framework technique, are presented. The same section is devoted to present recent related work on FTA. Section 3 presents the framework architectural description. SAFELAND and supported algorithms are discussed in sections 4 and 5, respectively. The paper concludes with our further orientations towards the integration of the framework with SAFELAND.

2. Background and related work

Fault Tree Analysis (FTA) is a form of safety analysis widely applied in aerospace, electronics and nuclear

[1]This work has been partially funded by the European Communities, under contract 20899.

industries. It was originally developed in 1961 by H. A. Watson at Bell Labs. to evaluate the Minuteman Launch Control System for an unauthorized (inadvertent) missile launch [12]. FTA is a top-down search technique used to prove that a given hazard (i.e., a set of conditions which is possible to lead to an accident [15]) is unlikely to arise. Analysis starts from a hazard and proceeds backwards looking for possible causes of that hazard.

There were several reasons that have led to the choice of FTA as the backbone analysis technique for the proposed framework. First of all, hazards associated with a particular safety critical application are usually well known; the developers have a list of known hazards which forms the basis for analyzing the application behavior. The set of hazards is unlikely to be modified, unless some new technology or a major change in operating philosophy takes place, indicating that new forms of an accident (or ways of causing an accident) are likely to be encountered. Therefore, the use of a top-down technique, such as FTA, for performing safety analysis seems to be appropriate.

Moreover, a structured approach is desirable. Although existing approaches for safety analysis are judgmental to some extent, any intervention of the safety analyst should be minimized for facilitating the automation of analysis activities. There are certain limitations associated with other safety analysis techniques. For example, HAZard and OPerability studies (HAZOP) [2] is usually applied to perform system safety analysis within the requirements stage, whereas attempts made for applying Failure Modes and Effects Analysis (FMEA) [15] to software have shown that software potentially has lots of failure modes and effects, making the use of FMEA inappropriate. Since our main intention is to define an, as much as possible, open and general framework for analyzing the safety of an application at requirements specification, design and development stages, FTA seems to be the most systematic technique.

Furthermore, FTA has been often proved to have much lower cost than conventional formal verification methods. For example, software FTA performed during the development of a nuclear power plant shutdown system [12] (which was about 3,000 lines of code written in Fortran and Pascal) took about three person-months (included the time required for analysts to become familiar with the technique). In contrast, the full formal verification of the same software using functional abstractions took about thirty person-years.

Work related to FTA has been successfully performed in the past, not only to the early stage of requirements specification, but also to the actual implementation. Leveson et al. in [16] examined the practicality of producing system level fault trees from a state machine model. In this work, they explored various types of analyses that can be performed on state machine models, including the generation of fault trees. These ideas can be adapted to many state machine models and, in particular, to the Requirements State Machine Language (RSML) [14].

The procedure of safety analysis on RSML specifications is based on a previous work by Leveson and Stolzy [11], related to the use of backward analysis for safety verification on Time Petri Net models. This procedure detects possible hazardous states that can be reached if the system operates correctly (i.e., it detects specification errors). The application of certain safety analysis procedures performed on Petri Nets to a more complex RSML model, as well as, the automated synthesis of fault trees are discussed in [16]. Automated fault tree generation from an RSML specification, uses a backward simulation in order to find configurations, such that there exists a set of transitions leading back to the current configuration.

However, when one applies FTA to design notations has to face the possibility that the produced code will not reflect the original design, and therefore, not provide adequate safety. When FTA is applied to source code, it can show the calling hierarchy and the interrelationships between the actual code modules and the top event of interest (hazard). Specifically, several benefits may arise from the application of FTA to software safety analysis [19]:

- it seems easier to mitigate potential problems, since code is a precise definition of exactly what computations are being executed
- there exist techniques directly applied to source code (e.g., the fault tree templates for the ADA language, described in [13])
- potential sources of hazards can be more easily decomposed and analyzed
- a fault tree, automatically generated from the code, can be compared to previous fault trees, generated from previous code versions, so as to trace and determine how "safer" the system becomes
- a fault tree, automatically generated from the code, can be compared to a fault tree manually generated from the design, to indicate whether the original intuition about what can or cannot lead to a hazard was accurate, and thus, substantiating or disproving the original safety hypotheses.

From all the above it is evident that the development of an integrated environment for applying safety analysis at requirements, design and implementation phases, is still an open issue. Toward this direction, in the next section, we propose the architecture of a framework which exploits the benefits of FTA, and can be potentially used as a generator of automated safety analysis tools.

3. Description of the framework

The proposed framework consists of four main modules: (1) the Parser, (2) the Fault Tree Constructor, (3) the Fault Tree Mitigator, and (4) the Hazard Analyzer. The framework architecture is depicted in Figure 1. Squares in the middle of the figure correspond to the four main modules. Squares with rounded corners indicate several types of information, while arcs represent flow of information between modules.

In particular, the framework operation follows four distinct phases:

Phase 1: a description (design or source code) of a safety critical application is provided as input to the parser module, which is the front end part of the whole scheme. This module is responsible to produce an intermediate format (i.e., a portable transformation of the description semantics) which serves as the basis for the subsequent phase.

Phase 2: the output of the parser module is passed to the Fault Tree Constructor, which uses simple fault tree templates (defined for the intermediate format) in order to produce a complicated fault tree template.

Phase 3: the produced fault tree will be mitigated and simplified by the Fault Tree Mitigator into a minimal cut-set form by using simple boolean algebra rules [3].

Phase 4: the resulted minimal fault tree will be combined with user defined hazardous conditions. Hazard Analyzer produces a safety report, showing the relationships among events specified in the hazard and events appeared in the input description. This report is the final output.

Obviously, the parser module is dependent on the programming language or the design notation that has been used for describing the input. The main scope of the parser is to identify within the input description primary events, conditions and generic logic statements, corresponding to primitives that appear in procedural programming languages (e.g., if-then-else, assignments, procedure and function calls), as well as, to produce the corresponding intermediate format.

Fault tree construction is a language independent phase that scans the intermediate code produced in the previous step. This module uses generic fault tree templates [13] and generates the corresponding fault tree. However, this module becomes semi-independent in case that input description contains specific features (e.g., source code containing special library function calls). Then, fault tree construction has to incorporate additional pre-described templates.

Fault tree mitigation is a completely language independent two-step phase. In the first step, the fault tree is converted into a minimal cut set form, while in the second step, any redundant information concerning events and conditions is eliminated [5, 19]. This procedure results in a three-level fault tree, where the root indicates an abstract event for the whole input, second-level nodes indicate intermediate events (pseudoevents) and leaf-nodes represent all primary events and conditions found in the input description. Intermediate nodes are connected to the root via an OR boolean gate, while leaf-nodes are connected to their parent pseudoevent via an AND gate.

Finally, Hazard Analysis phase checks, for each produced cut set, whether involved events and conditions can hold simultaneously with the user-defined hazardous condition. If this proposition is true for any cut set, the original input description should be considered as unsafe. Consequently, the developer has some clue of where an "error" may appear.

4. Case study: the SAFELAND tool

SAFELAND is the name of a safety analysis tool which follows the framework phases as they presented in the previous section. It performs software-based FTA according to a preliminary version of the whole framework as it was specified in [5].

Several considerations about the operation of SAFELAND with applications following different programming styles, guidelines and special primitives have been made in the context of OMI/ANTI-CRASH project, all related to the use of ANSI C in the implementation of safety critical real-time applications. So far, the following options have been encountered:

- *A parser oriented to the methodology used by the end-user partner (Thomson-CSF/TSI) during the development of safety critical systems* [18]. Having an actual application as a basis (such as the ServoValve braking system developed by Thomson-CSF/TSI in OMI/ANTI-CRASH), the parser and fault tree construction module should process ANSI C programs, a specific structured specification language called SAO (Specification Assistess Ordinateur), and the special guidelines for Thomson's software developers.
- *A parser oriented to the ANDF (Architectural Neutral Distribution Format).* According to the ANDF requirements and specifications [1], a parser of this kind should support a trustable subset of ANSI C that satisfies the RTCA/DO-178B standard [17] defined for the certification of real-time airborne systems software.
- *A parser oriented to both Thomson's methodology, for the development of safety critical real-time applications, and the ANDF compiler technology.* Experiences derived from the application of ANDF technology to hard-real time systems [1] have shown

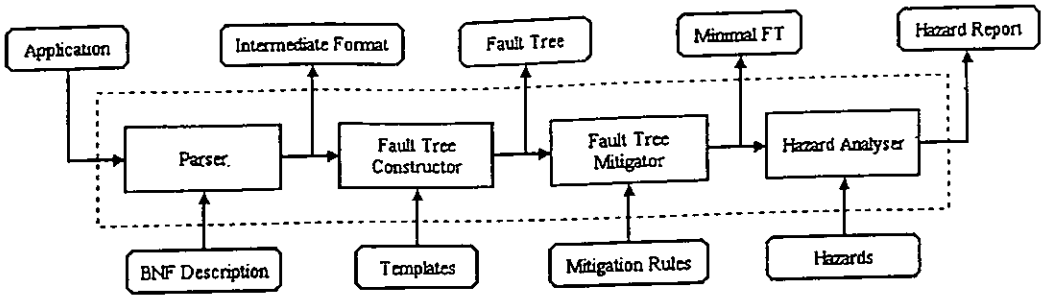

Figure 1: Framework architecture

that a parser of this kind should support -and take advantage of - all features appeared in both previous cases.

Currently, SAFELAND supports analysis on applications expressed in a subset of ANSI C and basic SAO library functions.

The SAO graphical design methodology [18] was defined in the context of the ESPRIT project 8034-OMI/FEM. It contains symbols that can be used for designing safety critical real-time applications. In particular, SAO codes almost all generic statements found in procedural programming languages and it is language independent. Thus, SAO designs can be easily converted to source code expressed in several programming languages. Software development based on SAO follows the "V" life cycle model (Figure 2). SAFELAND is a tool applied to applications developed using the SAO notation. Therefore, the tool can be applied during "Unit Tests" and "Integration Tests" phases. Any feedback to the implementation phase, after the verification of the existence of a possible hazard, guarantees that the application becomes safer with respect to the user defined hazards, without causing any violation to the "V" software development life cycle.

The ServoValve application is a specific real-time vetronics (vehicle-electronics) product used for avionics braking control. Considering the ServoValve braking application as a real case study, the following assertions can be listed:

- Every C module corresponds to an SAO module and contains the definition of a single function.
- For every SAO symbol, a special piece of code, which implements its functionality, exists.
- There are standardized SAO library routines incorporated into the program code.
- There are no complicated arithmetical operations. During the SAO design phase, a complicated arithmetical expression is calculated in many steps using appropriate SAO symbols. The corresponding C code just implements these symbols.
- There may be complicated boolean expressions. Although, during the SAO design phase, a complicated boolean expression is calculated in many sequential steps, the representation of these symbols in C code is usually implemented by a single complex boolean expression which corresponds to the specific sequence of SAO symbols.
- A function which implements an SAO module takes no parameters
- A function corresponding to an SAO module does not return any value (void), but it performs modifications to its environment via global variables.
- There are several functions that perform I/O, which are machine dependent. For example, in the application version for the ST9 processor (a microcontroller developed by SGS-THOMSON) there are functions that perform I/O from/to an EEPROM.

Furthermore, real-time applications written in C programming language and developed according to the SAO design methodology, satisfy certain requirements for safety critical real-time applications as they are described in RTCA/DO-178B standard [17]:

- pointers and dynamic data are not used
- recursion is forbidden and,
- use of infinite loops is limited only to the main module of the program (e.g., the control loop of the ServoValve in our case).

Following the requirements of RTCA/DO-178B standard, the Thomson's guidelines [18] (used during the development of vetronics hard real-time applications) simplify much more the application code. The main intention is to make the application safer (strictly deterministic). Some of these guidelines are listed below:

- pass of parameters between assembly and C routines has to be specified
- only standard features of C with guaranteed deterministic behavior are allowed
- data structure with dynamic length are discouraged
- explicit data conversions are not allowed
- all external and static memory has to be initialized
- specification of compilation mode is necessary
- nested macro definitions should not be used.

The SAFELAND parser both supports SAO library calls and follows Thomson's guidelines. The parser uses a reduced BNF specification of ANSI C, enriched by a subset of standardized SAO library functions (i.e., SAO arithmetic operations).

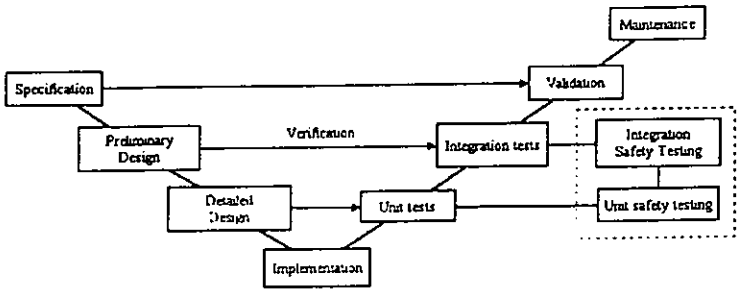

Figure 2 : "V" Life-cycle process

Currently, the main parser limitation is due to the fact that many routines of the ServoValve are written in the assembly language of the ST9 processor (in most cases, real time applications have parts of their code dependent on the target processor). Therefore, the corresponding SAO assembly low-level functions (besides arithmetic operations) are realized in SAFELAND as compound and no further analyzed events. In order to solve this problem the application of ANDF can be proved very efficient. In particular, two are the main goals of ANDF: to guarantee the portability of software, and to provide a complete and comprehensive concept for porting and distributing software from a development platform to a numerous different target computer architectures. Within the OMI/ANTI-CRASH project, the ANDF compiler technology is being enriched with new primitives based on the RTCA/DO-178B standard in order to become adequate for safety critical hard real-time applications. Although these primitives can guarantee some safety up to a point, they cannot eliminate hazardous conditions, as these conditions are part of the application semantics. SAFELAND does not currently address ANDF features, as related research in providing mechanisms that will satisfy requirements for safety critical hard real-time applications, is in progress.

5. SAFELAND algorithms

A demo version of the SAFELAND parser transforms small routines, coded in an ANSI C subset enriched with SAO library calls, into a simple intermediate format. The parser also extracts primary events and conditions from the input. A sample input routine for the parser is depicted in Figure 3. This routine is part of the ServoValve application. Without emphasizing to the actual meaning of the variables and constants, we will present an example of the performed safety analysis.

The intermediate format (pseudocode) consists of simple statement structures (e.g., *if condition then event, event is {event, event,...}*) mixed with SAO calls. Some of the events and conditions which appear in this example are listed in Table 1[2].

The Fault Tree Constructor processes the produced pseudocode by considering the fault tree templates defined for the generic statements of the intermediate format and the SAO library functions. We will show how the fault tree for the pseudocode is produced using the templates for *if-then* and *if-then-else* statements, *ADD* and *SUB* SAO library functions (i.e., *ADD* and *SUB* SAO functions perform addition and subtraction respectively, between two arithmetic variables, checking for overflow and underflow).

```
LIMIT ( )
{
  if (CONSIGNE > X_MAX)
        CONSIGNE = X_MAX;
  if (CONSIGNE < X_MIN)
        CONSIGNE = X_MIN;
  if (CONSIGNE >= CONS_PRES)
  {
        VARTEMP = SUB(CONSIGNE,CONS_PRES);
        if (VARTEMP > RL_UP)
              CONSIGNE=ADD(CONS_PRES,RL_UP);
  }
  else
  {
        VARTEMP = SUB (CONS_PRES,CONSIGNE);
        if (VARTEMP > RL_DOWN)
              CONSIGNE=SUB(CONS_PRES,RL_DOWN);
  }
  CONS_PRES = CONSIGNE;
}
```

Figure 3: Code of the LIMIT function

Initially, the whole pseudocode is considered as a compound event and a simple fault tree node is associated with it. At each step, the algorithm transforms every event either to the semantic denoted by the corresponding fault tree template or to the events which it consists of (in case of a compound event). More precisely, there may be either simple events or compound events nested in another compound event. The fault tree nodes which correspond to events that are contained in a compound event, are all located at the same level of the fault tree; these nodes are connected to the parent node which represents the compound event, via an AND gate. If an event is directly associated with a generic statement of the intermediate

[2]Conditions and events derived from the analysis of the SAO library functions ADD and SUB are omitted for the shake of simplicity.

Events		Conditions	
e1	CONSIGNE=X_MAX	c1	CONSIGNE>X_MAX
e2	CONSIGNE=X_MIN	c2	CONSIGNE<X_MIN
e3	VARTEMP=SUB(CONSIGNE,CONS_PRES)	c3	CONSIGNE>=CONS_PRES
e4	CONSIGNE=ADD(CONS_PRES,RL_UP)	c4	VARTEMP>RL_UP
e5	VARTEMP=SUB(CONS_PRES,CONSIGNE)	c5	VARTEMP>RL_DOWN
e6	CONSIGNE=SUB(CONS_PRES,RL_DOWN)		
e7	CONS_PRES=CONSIGNE		

Table 1: Basic events and conditions for the LIMIT function

format or an SAO library function, the boolean gates defined by the appropriate template will be inserted to the fault tree. The procedure is repeated recursively until there is no other event to be analyzed.

Based on this algorithm, Figure 5 presents the fault tree produced for the example code of Figure 3. However, this is not the final fault tree, as the templates of the SAO library functions (Figure 4) have not been added for the shake of simplicity.

By applying simple boolean algebra rules, the fault tree is converted to a minimal cut set form [3, 19]. Firstly, the fault tree is considered as a whole boolean expression and is reduced to an equivalent boolean sum of products (e.g., an expression in the form (A+B)(C+D) is converted to the equivalent AC+AD+BC+BD). Secondly, the produced sum of products is further simplified by applying boolean rules for simplification (e.g., ABC+AB is simplified to AB). Figure 6 shows the final fault tree for the example function in minimal cut set form.

Each cut set in the tree corresponds to a possible execution sequence and contains all conditions and events related to that sequence. For the whole ServoValve application, the procedure of hazard analysis requires the solution of a system of linear inequalities (i.e., each set of inequalities corresponds to set of events, conditions and hazards). We consider non-linear SAO functions as separate variables, i.e., the MUL symbol which indicates the multiplication of two variables A and B is considered as a separate variable A*B.

The hazard analysis algorithm considers a variable in different instances. In the final system of linear inequalities, each different instance of a variable corresponds to a distinct variable. A new instance of a variable is created when it appears in the left part of an assignment event. The approach does not increase the complexity of the algorithm as only one instance of a variable can be active at any inequality. The linear inequalities which correspond to a cut-set in the mitigated fault tree (presented in Figure 6) and a user-defined hazard, are all shown in Table 2. This example does not consider the special semantics of *ADD* and *SUB* SAO functions and treats them as simple addition and subtraction, respectively. Actually, this is a real case which results after the mitigation of the fault tree.

Concluding, SAFELAND has been proven efficient handling complexity issues inherent in FTA and related to the size of the produced fault trees. Applying the tool to the modules of the ServoValve application has revealed the following assertions:

- The nature of the analyzed code is very simple, it follows the guidelines of RTCA/DO-178B standard and methodologies for deterministic programming as previously mentioned.
- Each application module is analyzed separately; any call to another module is treated as a compound event in the produced fault tree, which, in sequence, will be analyzed in the respective fault tree structure.
- Conversion in minimal cut set form produces a mitigated fault tree (for example in Figure 6 the final mitigated fault tree for the LIMIT function of SVA contains only 16 cut sets).

6. Conclusions and future work

We believe that the adoption of the proposed framework for applying software safety analysis to safety critical real-time applications will not only help to detect hazardous conditions, but will also reduce costs during the software development cycle.

Framework characteristics such as generality and openness have already validated to a certain degree by the development of SAFELAND, a tool which follows the framework disciplines. SAFELAND has proved that the framework can support analysis on applications developed by using ANSI C and specification languages such as SAO.

Two are the main issues that we have to cope with in the near future. First, we have to analyze some safety critical functions of the testbed application (ServoValve) which are machine dependent. A possible solution is to allow some human assistance, while the parsing process is in progress. Second, SAFELAND has to be extended in order to support applications in C enriched with ANDF primitives. This problem will be encountered as soon as there are results demonstrating the performance of ANDF in real-time systems area.

Another future research and development scope is to make SAFELAND interoperate with SCAN, a

Schedulability Analysis Tool [6, 8], and MAT [7], a Monitoring Tool under development within the ESPRIT project 20576 - OMI/TOOLS. Currently, SCAN and MAT can be used in conjunction in order to analyze the dynamic behaviour of a real-time application. MAT provides actual application data (e.g., execution and blocking times). Consequently, these data along with user requirements are examined by SCAN to check the feasibility of timing constraints. The development of SAFELAND and the subsequent packaging of MAT, SCAN and SAFELAND in an integrated environment, will unify heterogeneous types (both static and dynamic) of software analysis. This way, a unified platform will be provided for the complete safety, reliability and performance analysis of real-time applications.

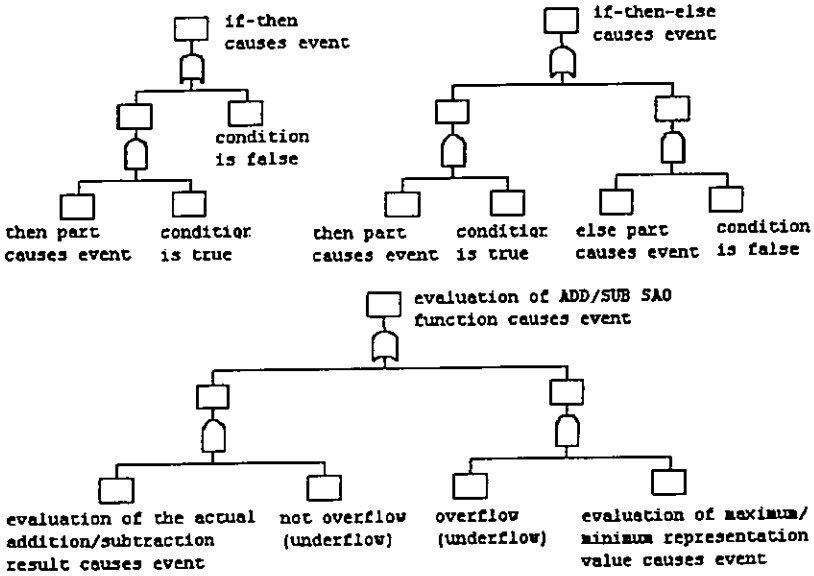

Figure 4: Fault Tree templates

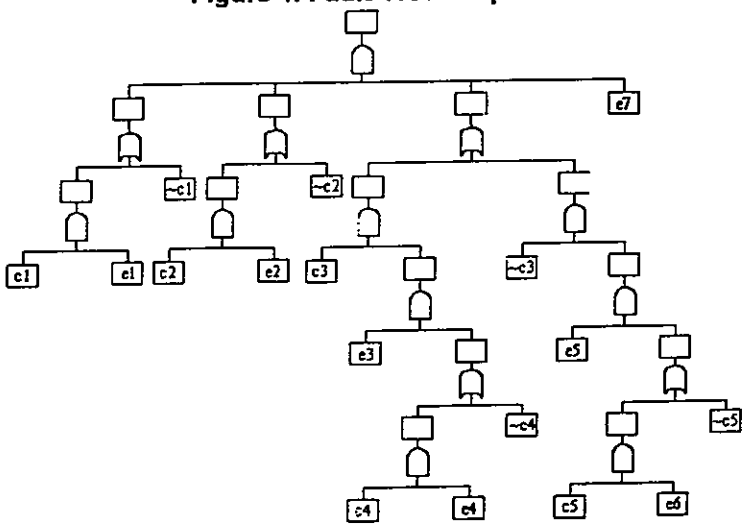

Figure 5: Fault Tree for the LIMIT function

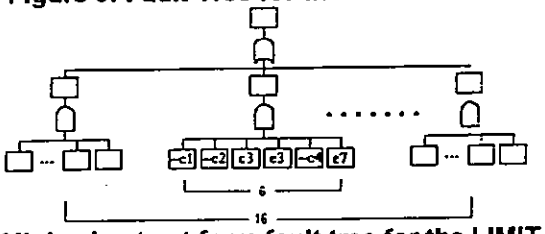

Figure 6: Minimal cut-set form fault tree for the LIMIT function

User Defined Hazard			Cut Set	
Mnemonic	Hazardous Condition		Mnemonic	Event/Condition
H	CONS_PRES=100 at the end of the LIMIT function		~c1	CONSIGNE<=X_MAX
	Constant	Value	~c2	CONSIGNE<=X_MIN
	RL_UP	1,280	c3	CONSIGNE>=CONS_PRES
	RL_DOWN	1,280	e3	VARTEMP=CONSIGNE-CONS_PRES
	X_MAX	32,767	~c4	VARTEMP<=RL_UP
	X_MIN	-32,767	e7	CONS_PRES=CONSIGNE
	The fact (CONSIGNE=100) and (-1180<=CONS_PRES<=100) leads to the user defined hazard			

Table 2: Hazard analysis of the LIMIT function

References

[1] H. Berlejung, W. Baron, "*Applying the ANDF Technology to Hard Real-Time Systems*". Embedded Microprocessor Systems, C. Muller-Schloer et al. (Eds.). IOS Press, 1996, pp. 420-429.

[2] E. J. Broomfield & P. W. H. Chung, "*Safety Assessment and Software Requirements Specification*". Technical Report, Chemical Engineering Department. Loughborough Univ. of Technology, UK.

[3] S. Contini, "*A New Hybrid Method for Fault Tree Analysis*". Reliability Engineering and System Safety, 49, 1995, pp. 13-21.

[4] ESPRIT project 20899 OMI/ANTI-CRASH, "*Technical Aneex*". September 1995.

[5] V. C. Gerogiannis, D. A. Brouxa & I. E. Caragiannis, "*Safety & Reliability Analysis Methodology Algorithms for Safety Critical Hard Real Time Systems*". ESPRIT project 20899 OMI/ ANTI-CRASH, D2.1. June 1996.

[6] V. C. Gerogiannis, M. A. Tsoukarellas. "*SAT-A Schedulability Analysis Tool for Real-Time Applications*". Proceedings of the 7th EUROMICRO Workshop on Real-Time Systems, IEEE Press, Odense, Denmark, June 14-16, 1995, pp. 155-161.

[7] V. C. Gerogiannis, M. A. Tsoukarellas et al., "Monitoring Tool - Requirements Analysis & Specification Report". ESPRIT project 20576 OMI/TOOLS, TR 5.1.1, August 1996.

[8] V. C. Gerogiannis & M. A. Tsoukarellas. "*Using SCAN to Analyze the Schedulability of a Real-Time Application*". Embedded Micro-processor Systems, C. Muller-Schloer et al. (Eds.), IOS Press, 1996, pp. 344-353.

[9] M. H. Klein, T. Ralya, B. Polak, R. Obenza & M. G. Harbour, "*A Practitioner's Handbook for Real-Time Analysis*". Carnegie Mellon University, Software Engineering Institute, Kluwer Academic Publishers, 1993.

[10] N. G. Leveson & P. R. Harvey, "*Analyzing Software Safety*". IEEE Transactions on Software Engineering, Vol. 9(5), September 1983, pp. 569-579.

[11] N. G. Leveson & J. L. Stolzy, "*Safety Analysis using Petri Nets*". IEEE Transactions on Software Engineering, Vol. 13(3), March 1987, pp. 386-397.

[12] N. G. Leveson, "*Safeware: System Safety and Computers*". Addison-Wesley, 1995.

[13] N. G. Leveson, S. S. Cha & T. S. Shimeall, "*Safety Verification of Ada Programs using Software Fault Trees*". IEEE Software 6(4), July 1991, pp. 48-59.

[14] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth & J. D. Reese, "*Requirements Specifications for Process Control Systems*". IEEE Transactions on Software Engineering, 20(9), September 1994, pp. 684-707.

[15] J. A. McDermid, "*Safety Engineering and Assurance for Real-Time Systems*". In Real Time Computing, W. A. Halang & A. D. Stoyenko (Eds.), NATO ASI Series, series F: Computer Systems and Sciences, Vol. 127, Springer-Verlag, 1994, pp. 131-160.

[16] V. Ratan, K. Partridge, J. D. Reese & N. G. Leveson, "*Safety Analysis Tools for Requirements Specification*". COMPASS 96, Gaithersbung, Maryland, 1996.

[17] Requirements and Technical Concepts for Aviation. "*RTCA/DO-178B: Software Consideration of Airborne Systems and Equipment Certification*". RTCA Inc., Washington, DC, December 1992.

[18] "*Regles de Conception et de Codage*". Thomson-CSF/DOI, 1991.

[19] J. M. Voas & K. W. Miller, "*An Automated Code-based Fault Tree Mitigation Technique*". Tech. Report, Reliable Software Technologies Corporation.