Integrating Fault Tree Analysis with Event Ordering Information [*]

Marco Bozzano & Adolfo Villafiorita
ITC - IRST, Via Sommarive 18, Povo, 38050 Trento, Italy
{bozzano,adolfo}@irst.itc.it

ABSTRACT: Fault tree analysis is a traditional and well-established technique for analyzing system design and robustness. Its purpose is to identify sets of basic events, called *cut sets*, which can cause a given *top level event*, e.g., a system malfunction, to occur. In this paper we present an algorithm that extracts *ordering* information, i.e., finds out possible ordering constraints which are required to hold between basic events in a cut set. The algorithm is completely *automatic*, and has been incorporated into a more general framework, based on model checking techniques, for automatic fault tree generation and analysis.

1 INTRODUCTION

The development of safety critical systems requires to check that the system behaves as expected not only in nominal situations, but also under certain degraded situations. Thus, on the one hand, system models are developed by the design engineers in order to specify and to analyze the expected behaviour of the system under consideration. On the other hand, the envisaged system is analyzed by safety specialists with respect to malfunctions, i.e., unintended behaviour. The safety analysis, performed at each stage of the system development, is intended to identify all possible hazards with their relevant causes. Traditional safety analysis methods include, e.g., Functional Hazard Analysis, Failure Mode and Effect Analysis (FMEAs), and Fault Tree Analysis (FTA) (Vesely et al. 1981).

Fault tree analysis (Vesely et al. 1981), in particular, is a deductive and top-down method to analyze system design and robustness. Roughly speaking, the FTA process consists in picking a *top level event* (e.g., a system malfunction condition) and identifying all possible *sets* of basic events, called *cut sets*, which can cause the top event to occur. Among them, one would like to isolate *minimal* cut sets, that is, cut sets which do not include events that ultimately do not affect the occurrence of the top event. The information on cut sets is then collected in a *fault tree*, which consists of system and component events, connected by *gates* which define the logical relations between events. The *cut set* representation provided by traditional fault tree analysis is not structured. A cut set is simply seen as a flat collection of basic events, and no information is provided about their mutual relationship. Although events are often allowed to happen in any order, in general there may be *timing constraints* which enforce a particular event to happen before or after another one. This can happen as a result of a causality relation, a functional dependency, or more subtle reasons related to dynamic scenarios where system behaviour can be affected by, e.g., automatic control systems or operator actions (Siu 1994).

In this paper, we are interested in *automatically* computing ordering information of basic events. Specifically, given a top level event and a minimal cut set computed via fault tree analysis, we want to find out whether there are *ordering constraints* which hold between pairs of basic events in the cut set. We call this *event ordering analysis*. We present an algorithm which integrates traditional fault tree analysis by providing event ordering information for basic events in a cut set. The algorithm is completely *automatic*, and has been incorporated into a more general framework for automatic fault tree generation and analysis. The core of our ordering analysis algorithm is based on known procedures for *minimization* (i.e., computation of *minimal cut sets*) of *boolean functions* (Coudert and Madre 1992; Coudert and Madre 1993; Manquinho et al. 1998) represented as Binary Decision Diagrams (BDDs) (Bryant 1992). The encoding of the problem and some adjustments necessary to deal with *inconsistency* are original. The encod-

[*]This work has been and is being developed within ESACS, an European sponsored project, contract no. G4RD-CT-2000-00361. See also URL http://www.esacs.org/

ing is based on ordering information variables, that is, variables which relate pairs of different basic events, tracking the information about the mutual order in which the two events may or may not occur.

Our framework is based on model checking (Clarke et al. 2000), a well-established method for formally verifying temporal properties of finite-state concurrent systems. Model checking has been applied for the formal verification of a number of significant safety-critical industrial systems (Holzmann 1997; Larsen et al. 1997; Cimatti et al. 2002). We have incorporated fault tree and ordering analysis functionalities into the model checking tool NuSMV (Cimatti et al. 2002), a BDD-based symbolic model-checker developed at ITC-IRST, originated from a re-engineering and re-implementation of SMV (McMillan 1993). NuSMV is a well-structured, open, flexible and well-documented platform for model checking, and it has been designed to be robust and close to industrial system standards (Cimatti et al. 2000).

This line of research has been carried on inside the ESACS project, an European-Union-sponsored project whose main goals are to define a methodology to improve the safety analysis practice for complex systems development, to set up a shared environment based on tools supporting the methodology, and to validate the methodology through its application to case studies. The fault tree and ordering analysis functionalities which we discuss in this paper have been included in a more general safety analysis platform which we are developing inside the ESACS project (Bozzano and al. 2003).

Structure of the paper. The rest of the paper is structured as follows. In Section 2 we give a brief overview of the basics of fault tree analysis and we introduce *event ordering analysis*, explaining its significance and its relationship with model checking. In Section 3 we introduce a simple example which we will use in Section 4, where we present our minimization algorithm for ordering analysis and we briefly discuss its integration with fault tree analysis based on model checking. Finally, in Section 5 we discuss related work and draw some conclusions.

2 EVENT ORDERING ANALYSIS

Fault Tree Analysis (FTA) (Vesely et al. 1981; Liggesmeyer and Rothfelder 1998; Rae 2000) is a deductive, top-down method to analyze system design and robustness. It usually involves specifying a *top level event* (TLE hereafter) to be analyzed (e.g., a *failure state*), and identifying all possible sets of basic events (e.g., basic *faults*) which may cause that TLE to occur. Benefits of FTA include, e.g.: identify possible system reliability or safety problems at design time; assess system reliability or safety during operation; identify root causes of equipment failures. *Fault trees* provide a convenient symbolic representation of the combination of events resulting in the occurrence of the top event. Fault trees are usually represented in a graphical way, structured as a parallel or sequential combination of AND/OR gates.

In this paper we are interested in deductive methods which can be used to automatically generate fault trees starting from a given system model and top level event. In particular, we focus on analysis techniques based on model checking. Model checking (Clarke et al. 2000) is a well-established method for formally verifying temporal properties of finite-state concurrent systems. System specifications are written as temporal logic formulas, and efficient symbolic algorithms (based on data structures like BDDs (Bryant 1992)) are used to traverse the model defined by the system and check if the specification holds or not. The application of model checking to fault tree generation works in the following way. Given a system model and a top level event (TLE) to analyze, model checking techniques can be used to extract *automatically* all collections of basic events (called *minimal cut sets*) which can trigger the TLE. The generated cut sets are minimal in the sense that only events that are strictly necessary for the TLE to occur are retained.

In this paper, we discuss and propose an algorithm for extending FTA with *event ordering information*. In traditional FTA, cut sets are simply flat collections (i.e, conjunctions) of events which can trigger a given TLE. However, there might be timing constraints enforcing a particular event to happen before or after another one, in order for the TLE to be triggered (i.e., the TLE would not show if the order of the two events were swapped). Ordering constraints can be due, e.g., to a causality relation or a functional dependency between events, or caused by more complex interactions involving the dynamics of the system under consideration. Whatever the reason, event ordering analysis can provide useful information which can be used by the design and safety engineers to fully understand the ultimate causes of a given system malfunction, so that adequate countermeasures can be taken.

3 AN EXAMPLE

We present below an example which we will use in Section 4 to explain our methodology. The example is deliberately simple for illustration purposes and should not be regarded as modeling a realistic system. We refer to (Bozzano and al. 2003) for more meaningful examples to which the methodology and the algorithm have been applied. Let us consider the circuit drawn in Figure 1. The circuit is composed of two JK flip-flops and an OR gate, and it has three input bits and one output. In short, a JK flip flop is a (clock driven) logical component with two input bits ('J' and 'K') and two output bits ('Q' and '!Q', the latter sim-

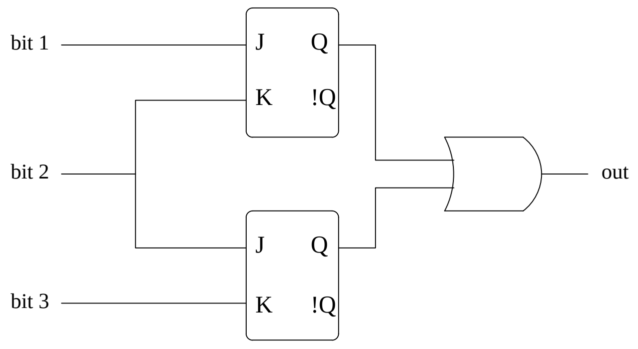

Figure 1: A simple circuit with two JK flip-flops

ply being the negation of the former). The truth table of the JK flip flop is such that whenever 'J' and 'K' are low the output signal 'Q' (which can be either low or high) remains unchanged, whenever either 'J' or 'K' is set to high the output 'Q' is set to, respectively, one or zero, and, finally, a high signal on both 'J' and 'K' is used to *toggle* the current value of 'Q'.

In the circuit drawn in Figure 1, the three input bits are set to zero, but they can non-deterministically fail, at any time and in any order, in which case their value is inverted (i.e., it is set to one) forever (note that we assume *persistent* failures). Initially, we assume all signals to be low, i.e., the input bits, the 'Q' outputs of the flip-flops and, consequently, the output of the circuit, are all set to zero. A NuSMV model of the circuit is shown in Figure 2. It is composed of three modules, one for modeling an input bit, one for modeling a JK flip-flop (note that this module simply implements the truth- table of a JK flip-flop) and the main module, which puts all the components together and defines the output signal of the circuit. For simplicity, we have not modeled flip-flop clocks explicitly. We assume that the input values 'J' and 'Q' are transferred to the flip-flop outputs at each NuSMV transition (i.e., a NuSMV transition can be thought of as causing a triggering edge of the clock pulse).

Top level events to be used for fault tree analysis can be expressed in the temporal logic CTL (Emerson 1990). Arbitrary CTL formulas can be used to perform FTA in NuSMV. Some examples are:

$$\mathtt{AG(out)} \qquad (T_1)$$
$$\mathtt{AG((out} \rightarrow \mathtt{AX(!out))\ \&\ (!out} \rightarrow \mathtt{AX\ out))} \qquad (T_2)$$
$$\mathtt{EG((out} \rightarrow \mathtt{AX(!out))\ \&\ (!out} \rightarrow \mathtt{AX\ out))} \qquad (T_3)$$

The top level event T_1 is a CTL formula specifying all the states of the system in which the output of the circuit is *forced* to be set to value one *forever*, i.e., for *every* possible path (evolution of the system) the output is *globally* set to value one on that path. Similarly, the CTL formula T_2 is a specification of all the states such that the output of the circuit is *forced* to *oscillate* forever back and forth between the values zero and one. Finally, the CTL formula T_3 is a specification of all the states such that there exists *one* path on which the output is *globally forced* to oscillate forever.

As mentioned in Section 2, we have implemented a procedure for performing fault tree analysis in NuSMV. As an aside, we mention that the safety analysis platform we are developing inside the ESACS project provides additional features for managing failure modes. Specifically, fault tree computation starts with the user assigning a set of failures to the various components, which are then automatically inserted into the original model of the system. The result is an *extended system model* with failure variables (e.g., the variable `FailureMode` of the `bit` module in Figure 2). Model checking techniques can then be applied to the extended NuSMV model (e.g., the model in Figure 2) to extract *automatically* all collections of basic events, i.e., all *minimal cut sets*, which can cause any of the above TLEs. Cut sets are expressed in terms of *sets* of *failure events*, i.e., pairs consisting of a failure variable and a failure mode. The results of fault tree analysis for the model in Figure 2 and the CTL formulas T_1, T_2, and T_3 are shown below (hereafter, we shorten $(\mathtt{bit}i\mathtt{.FailureMode,inverted})$ with $\mathtt{bit}i\mathtt{\_inv}$). In this particular case, exactly one minimal cut set M_i is computed for each formula T_i (note that in general more than one cut set can be computed for a TLE).

$$\{\mathtt{bit1\_inv},\ \mathtt{bit3\_inv}\} \qquad (M_1)$$
$$\{\mathtt{bit1\_inv},\ \mathtt{bit2\_inv},\ \mathtt{bit3\_inv}\} \qquad (M_2)$$
$$\{\mathtt{bit2\_inv}\} \qquad (M_3)$$

For M_1, we have that, in order for the output of the circuit to be *forced* to return value one forever (property T_1), it is necessary that both the first and the third bit fail. Notice that the output of the circuit can also get stuck at value one as a result of a failure of the first bit only. In this case, however, the output of the circuit is not *forced* to that value, i.e., as the reader can verify, there exist possible evolutions of the circuit such that the output can assume value zero. In

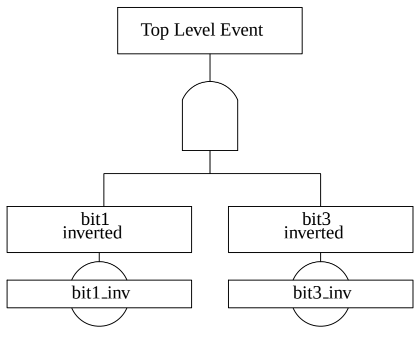

Figure 3: A Fault Tree for T_1

Figure 3 we show a simple graphical representation for the fault tree corresponding to T_1. The cut set is minimal in the sense that only events that are strictly necessary for the TLE to occur are retained. Similarly, minimal cut set M_2 states that all bits must be failed in order for the output of the circuit to be forced to oscillate forever. Notice that failure of all input bits is not a *sufficient* condition for oscillation of the circuit.

```
MODULE bit(input)
VAR
  out         : boolean;
  FailureMode : {no_failure,inverted};

ASSIGN
  init(FailureMode) := no_failure;
  next(FailureMode) := case
        FailureMode = no_failure : {no_failure,inverted};
        FailureMode = inverted   : inverted;
  esac;
  out := case
        FailureMode = no_failure : input;
        FailureMode = inverted   : !input;
  esac;
```

```
MODULE ff(J,K)
VAR
  Q : boolean;

ASSIGN
  init(Q) := 0;
  next(Q) := case
        !J & !K :  Q;
        !J &  K :  0;
         J & !K :  1;
         J &  K : !Q;
  esac;
```

```
MODULE main
VAR
  bit1   : bit(0);
  bit2   : bit(0);
  bit3   : bit(0);
  ff1    : ff(bit1.out,bit2.out);
  ff2    : ff(bit2.out,bit3.out);
  out    : boolean;

ASSIGN
  out := ff1.Q | ff2.Q;
```

Figure 2: A NuSMV model for the circuit in Figure 1

In fact, there are some *timing constraints* which must be satisfied in order for the circuit to show this oscillating behaviour. Extracting information about these timing constraints is exactly the purpose of the *ordering analysis* which is described in the next section.

4 THE MINIMIZATION ALGORITHM

In this section we explain in detail our algorithm for event ordering analysis. Specifically, we describe a procedure which takes in input a system model (e.g., the NuSMV model in Figure 2) and a cut set, and is able to extract event ordering information. The core of the algorithm is based on procedures for computing *prime implicants* of boolean functions (Coudert and Madre 1993), and exploits the BDD-based representation for boolean functions, which is used extensively in NuSMV. The algorithm is made up of a number of different phases, which are detailed below.

0) Pick a minimal cut set. A prerequisite of our procedure is having a system model SM and a top level event TLE at hand. Then, as explained in Section 3, we run NuSMV on SM and TLE, and we get a collection of minimal cut sets. Assuming the collection is not empty, we pick one MCS. The purpose of the ordering analysis algorithm is to extract ordering constraint information from MCS.

1) Generate the ordering information model. Assuming MCS is composed of a set of n failure events, say $(fm_1, var\_name_1), \ldots, (fm_n, var\_name_n)$, for each pair of distinct failure events with indexes i and j in MCS ($i \neq j$), we introduce a new ordering variable $\mathtt{order\_var\_name_{ij}}$ in the NuSMV model SM, to keep track of the mutual order in which the two failure events may happen. In order to give a complete encoding for ordering information, we thus need a total of $\frac{1}{2}n(n-1)$ ordering variables. We call the resulting model *ordering information model* (OIM hereafter). The NuSMV skeleton for defining an ordering variable is shown in Figure 4. The skeleton is instantiated each time with different actual parameters for failure variables and failure modes. Every ordering variable can assume one among the three values $\{\mathtt{before}, \mathtt{after}, \mathtt{simult}\}$, the intuition being that the first event happens before, after, or at the same time with the second one (note that the notion of simultaneousness is relative to the granularity of the NuSMV *step*, e.g., one clock pulse for the circuit in Figure 1). An auxiliary variable definition (which may assume the additional value `unknown`) is necessary in order to code the fact that the value of a given variable is still unknown during the computation (the `unknown` value will be eventually overwritten, because all failure events in MCS are forced to occur, see below).

2) Re-run NuSMV on the ordering information model. In this phase NuSMV is re-run on the OIM, with the same TLE, in order to track the information captured by the ordering variables. The analysis is specialized to the given MCS, i.e., the formula provided to NuSMV for the analysis forces the failure events contained in MCS (*and only them*) to occur. The result is a BDD representing all the different

```
VAR
  ORDER_VAR_NAME_AUX : {UNKNOWN, BEFORE, AFTER, SIMULT};
  ORDER_VAR_NAME : {BEFORE, AFTER, SIMULT};

ASSIGN
  init(ORDER_VAR_NAME_AUX) := UNKNOWN;
  next(ORDER_VAR_NAME_AUX) := case
        ORDER_VAR_NAME_AUX = UNKNOWN : case
                (VAR_NAME_1 = FM_1 & VAR_NAME_2 = NO_FAILURE) : BEFORE;
                (VAR_NAME_1 = NO_FAILURE & VAR_NAME_2 = FM_2) : AFTER;
                (VAR_NAME_1 = FM_1 & VAR_NAME_2 = FM_2)       : SIMULT;
                1                                             : UNKNOWN;
                esac;
        1                            : ORDER_VAR_NAME_AUX;
        esac;
  ORDER_VAR_NAME := case
        ORDER_VAR_NAME_AUX = UNKNOWN : SIMULT;  -- does not matter
        1                            : ORDER_VAR_NAME_AUX;
        esac;
```

Figure 4: NuSMV code skeleton for ordering variable definition

configurations (including system variables and ordering variables) which can cause TLE in presence of the failure events in MCS. For instance, consider the NuSMV model, the top level event T_2, and the relevant minimal cut set M_2 described in Section 3. The minimal cut set and the top level event are combined together, yielding the CTL formula

$$((\texttt{AG}\ (\texttt{out}) \rightarrow \texttt{AX}\ (!\texttt{out})) \ \&\ (!\texttt{out} \rightarrow \texttt{AX}\ \texttt{out})) \ \&\ \texttt{bit1\_inv} \ \&\ \texttt{bit2\_inv} \ \&\texttt{bit3\_inv}$$

NuSMV is fed with this formula in order to generate a BDD representing all states causing T_2 because of the failure events in M_2. This BDD also includes the information about ordering variables.

3) Abstract away non-ordering variables. In this phase we simply abstract away variables other than ordering ones. The result is still a BDD representing all the possible failure event orderings.

4) Extract ordering constraints. This phase contains the core of the minimization algorithm, and is composed of three interrelated sub-phases.

4.1. Add inconsistent configurations. The ordering variable encoding described in point 1 above is redundant in the following sense. Consider, e.g., three variables v_{ij}, v_{jk}, and v_{ik}, representing the order in which failure events i and j (j and k, i and k) occur. Clearly, if, say, v_{ij} and v_{jk} are both set to the value `before`, for *transitivity* also v_{ik} will be (necessarily) set to `before`. In other words, the encoding allows for *inconsistent* configurations which will never be the result of the model checking analysis (e.g., $\langle \texttt{before}, \texttt{before}, \texttt{after} \rangle$ in the previous example). During this phase we extend the BDD resulting from phase 3 with a BDD representing such inconsistent configurations (i.e., we consider their disjunction). This amounts to admitting inconsistent configurations as if they were perfectly legal. In this way (and with a little adjustment explained in phase 4.3 below) we can ensure that the minimization algorithm (see phase 4.2) works properly. Intuitively, the minimization algorithm works by picking variables whose value is *irrelevant* (in the sense that they can assume any among the possible *legal* values). Therefore, phase 4.1 is necessary to give the minimization algorithm enough information to correctly recognize which variables are irrelevant and which are not.

4.2. Compute prime implicants. We simply run the standard algorithm for computing *prime implicants* of a boolean function (Coudert and Madre 1993) on the BDD resulting after phase 4.1. For instance, for the top level event T_2 and cut set M_2 this phase computes 16 prime implicants. An example of prime implicant is the following (it represents the timing constraint enforcing the first bit to fail before the second one and the second one before the third one):

```
p0) -------------------------
    bit1_inv  **before**  bit2_inv
    bit2_inv  **before**  bit3_inv
```

4.3. Run simplification subroutine. The purpose of this phase is to cut *inconsistent* results and *subsumed* (i.e., logically implied) results from the output of phase 4.2. Inconsistent results can arise as a side effect of phase 4.1, and must be discarded after the minimization phase, whereas the purpose of the simplification subroutine is to retain only *minimal* results. For the top level event T_2 and cut set M_2, the previously generated 16 implicants are reduced to 3 after the simplification phase. For instance, the prime implicant `p0` is discarded because it is subsumed by prime implicant `p2` (see below).

5) Show results. The final output consists of the collection of prime implicants resulting after phase 4. For the top level event T_2 and cut set M_2, NuSMV outputs the following three prime implicants:

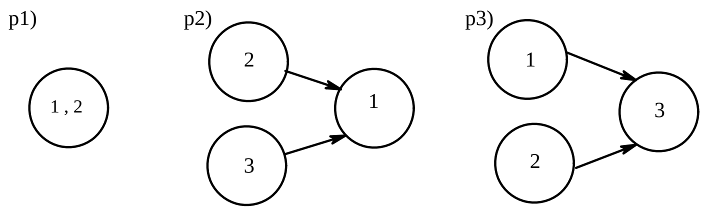

Figure 5: Prime implicants as precedence graphs

```
p1) ------------------------
    bit1_inv  **simult**  bit2_inv
p2) ------------------------
    bit2_inv  **before**  bit1_inv
    bit3_inv  **before**  bit1_inv
p3) ------------------------
    bit1_inv  **before**  bit3_inv
    bit2_inv  **before**  bit3_inv
```

Each prime implicant is a list of ordering constraints between failure events, and represents a different alternative (i.e., a different ordering possibly causing the TLE). Each list of failure events represents a *precedence graph*, showing which failure event must happen before which, for a given prime implicant. The output might be drawn in a more suggestive way, as shown in Figure 5. Each node in the graph contains one or more failure events (an index i denotes failure event bit$i$_inv), which are supposed to happen simultaneously, whereas each arrow represents a *before* relation between two (sets of) failure events.

The precedence graphs shown in Figure 5 represent the ordering information which has been obtained by running our algorithm on the model, top level event T_2 and minimal cut set M_2 described in Section 3. As discussed in Section 3, the CTL formula T_2 is a specification of all the states such that the output of the circuit is *forced* to oscillate forever between the values zero and one, whereas minimal cut set M_2 shows that all three input bits of the circuit must fail in order for such a behaviour to be observable. The ordering analysis results give us further information about this oscillating phenomenon, i.e, show that some further *timing constraints* between the failure events must hold as well. In particular, one of the orderings shown in Figure 5 must hold: either the first and the second bit must fail simultaneously (and the third one at any time, including simultaneously with the other two), or the second and third bit must fail before the first (the order between the former two bits is left unspecified), or the first and second must fail before the third (again, the order between the former two bits is left unspecified). A possible ordering which is *ruled out* by the results of our ordering analysis is the one in which the three bits fail in the following order: the first bit, then the third one, and finally the second one.

4.1 *Ordering and fault tree analysis*

We conclude this section with a brief explanation about how the minimization algorithm can be integrated with fault tree analysis. A system model is assumed to be given. The verification process consists of the following phases. First of all, a top level event to analyze is chosen (clearly, the analysis can be repeated for different top level events). Then, we run the minimization algorithm of (Coudert and Madre 1993) to compute the *minimal cut sets* of basic events causing TLE. Finally, for each cut set we generate an ordering information model and we perform the ordering analysis. The output is a fault tree for TLE, where each cut set in the tree is equipped with ordering information (that is, a precedence graph).

For the example of Section 3, the analysis can proceed in the following way. First, a top level event is chosen (e.g., T_1, T_2, or T_3) and NuSMV is run to perform fault tree analysis. The result is a collection of minimal cut sets of failure events. For each cut set, ordering analysis is performed on a suitable ordering information model. The results show that: for T_1 there are no timing constraints (all orderings between the two failure events are possible); for T_2, the output of NuSMV is shown in Figure 5; finally, T_3 only includes one basic event, hence it is useless to perform ordering information analysis on it.

5 CONCLUSIONS AND RELATED WORK

In this paper we have presented an algorithm which improves the results given by traditional fault tree analysis (Vesely et al. 1981). In particular, our algorithm performs what we call *event ordering analysis*. This analysis can be conveniently used to extract possible ordering constraints holding between basic events in a given cut set, thus providing a deeper insight into the causes of system malfunction and supporting the reliability and safety analysis process. Although very simple, the example in Section 3 shows the importance and significance of event ordering information. It also suggests that timing constraints can arise very naturally in industrial systems.

As explained in the paper, our algorithm for ordering analysis is based on classical procedures for *minimization* of *boolean functions*, specifically on the implicit-search procedure described in (Coudert and Madre 1992; Coudert and Madre 1993), which is based on Binary Decision Diagrams (BDDs) (Bryant 1992). This choice was quite natural, given that the NuSMV model checker makes a pervasive use of BDD data structures. For alternative explicit-search and SAT-based techniques for computation of prime implicants, see (Manquinho et al. 1998). The results computed by the algorithm may differ depending on the exact order in which ordering variables are chosen in the minimization step. This a consequence of the non-determinism which is inherent in the prime implicant computation (Coudert and Madre 1993). However, we conjecture that the minimization procedure enjoys some *optimality* properties which we are studying as part of our future work.

A large amount of work has been done in the area of probabilistic safety assessment (PSA) and in particular on *dynamic reliability* (Siu 1994). Dynamic reliability is concerned with extending the classical event or fault tree approaches to PSA by taking into consideration the mutual interactions between the hardware components of a plant and the physical evolution of its process variables (Marseguerra et al. 1998). Examples of scenarios which dynamic reliability tries to take into consideration are, e.g., human intervention, expert judgment, the role of control/protection systems, the so-called failures *on demand* (i.e., failure of a component to intervene), and also the ordering of events during accident propagation (Senni, Semenza, and Galvagni 1991; Cacciabue and Cojazzi 1994; Cacciabue and Cojazzi 1995). Different approaches to dynamic reliability include, e.g., state transitions or Markov models (Aldemir 1987; Papazoglou 1994), the dynamic event tree methodology (Cojazzi et al. 1992), and direct simulation via Monte Carlo analysis (Smidts and Devooght 1992; Marseguerra et al. 1998). The work which is probably closer to ours is (Cojazzi et al. 1992), which describes dynamic event trees as a convenient means to represent the timing and order of intervention of a plant sub-systems and their eventual failures. With respect to the classification the authors propose, our approach can support *simultaneous* failures, whereas, at the moment, we are working under the hypothesis of *persistent* failures (i.e., no repair is possible).

The most notable difference between our approach and the works on dynamic reliability mentioned above is that we present *automatic* techniques, based on model checking, for both fault tree generation and ordering analysis, whereas traditional works on dynamic reliability rely on manual analysis (e.g., Markovian analysis (Papazoglou 1994)) or simulation (e.g., Monte Carlo simulation (Marseguerra et al. 1998), the TRETA package of (Cojazzi et al. 1992)). Automation is clearly a point in favour of our framework. Furthermore, we support automatic verification of arbitrary temporal CTL properties (in particular, both safety and liveness properties). Current work is focusing on a number of improvements and extensions in order to make the methodology competitive with existing approaches and usable in realistic scenarios. First of all, there are some improvements at the modeling level. The NuSMV models used so far are discrete, finite-state transition models. In order to allow for more realistic models, we are considering an extension of NuSMV with hybrid dynamics, along the lines of (Henzinger 1996; Henzinger et al. 1997). This would allow both to model more complex variable dynamics, and also a more realistic modeling of time (currently, time is modeled by an abstract transition step). Furthermore, we need to extend our framework in order to deal with *probabilistic* assessment. Although not illustrated in this paper, associating probabilistic estimates to basic events and evaluating the resulting fault trees is straightforward. However, more work needs to be done in order to support more complex probabilistic dynamics (see, e.g., (Devooght and Smidts 1994)). Also, we want to overcome the current limitation to permanent failures.

We also mention (Manian et al. 1998; Sullivan et al. 1999), which describe DIFTree, a methodology supporting (however, still at the manual level) fault tree construction and allowing for different kinds of analyses of sub-trees (e.g., Markovian or Monte Carlo simulation for dynamic ones, and BDD-based evaluation for static ones). The notation the authors use for non-logical (dynamic) gates of fault trees and the support for sample probabilistic distributions could be nice features to be integrated in our framework.

Finally, the line of research concerning ordering analysis has been carried out inside the ESACS project. As a contribution to the project, we are developing an integrated platform providing the safety engineers with tools for the specification, analysis and validation of complex systems. Formal verification functionalities of the platform are based on model checking, and in particular on NuSMV(Cimatti et al. 2002). In this paper we have focused only on the aspects related to the minimization procedure. We refer the reader to (Bozzano and al. 2003), where a more detailed description of the project goals, the ESACS methodology, and more realistic examples to which the methodology has been applied can be found.

REFERENCES

Aldemir, T. (1987). Computer-assisted Markov Failure Modeling of Process Control Systems. *IEEE Transactions on Reliability R-36*, 133–144.

Bozzano, M. and al. (2003). ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *Proc. of European Safety and Reliability Conference (ESREL'03)*.

Bryant, R. (1992). Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys 24*(3), 293–318.

Cacciabue, P. and G. Cojazzi (1994). A human factor methodology for safety assessment based on the Dylam approach. *Reliability Engineering and System Safety 45*, 127–138.

Cacciabue, P. and G. Cojazzi (1995). An integrated simulation approach for the analysis of pilot-aeroplane interaction. *Control Engineering Practice 3*(2), 257–266.

Cimatti, A., E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella (2002). NuSMV2: An OpenSource Tool for Symbolic Model Checking. In *Proc. 14th International Conference on Computer Aided Verification (CAV'02)*, LNCS. Springer-Verlag.

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

Clarke, E., O. Grumberg, and D. Peled (2000). *Model Checking*. MIT Press.

Cojazzi, G., J. M. Izquierdo, E. Meléndez, and M. S. Perea (1992). The Reliability and Safety Assessment of Protection Systems by the Use of Dynamic Event Trees. The DYLAM-TRETA Package. In *Proc. XVIII Annual Meeting Spanish Nucl. Soc.*

Coudert, O. and J. Madre (1992). Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. 29th Design Automation Conference (DAC'98)*, pp. 36–39. IEEE Computer Society Press.

Coudert, O. and J. Madre (1993). Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In *Proc. Annual Reliability and Maintainability Symposium*.

Devooght, J. and C. Smidts (1994). Probabilistic Dynamics; The Mathematical and Computing Problems Ahead. In T. Aldemir, N. O. Siu, A. Mosleh, P. C. Cacciabue, and B. G. Göktepe (Eds.), *Reliability and Safety Assessment of Dynamic Process Systems*, Volume 120 of *NATO ASI Series F*, pp. 85–100. Springer-Verlag.

Emerson, E. (1990). Temporal and modal logic. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science*, Volume B, pp. 995–1072. Elsevier Science.

Henzinger, T. A. (1996). The Theory of Hybrid Automata. In *Proc. 11th Annual International Symposium on Logic in Computer Science (LICS'96)*, pp. 278–292. IEEE Computer Society Press.

Henzinger, T. A., P.-H. Ho, and H. Wong-Toi (1997). HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer 1*, 110–122.

Holzmann, G. (1997). The Model Checker SPIN. *IEEE Transactions on Software Engineering 23*(5), 279–295.

Larsen, K., P. Pettersson, and W. Yi (1997). UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer 1*(1-2), 134–152.

Liggesmeyer, P. and M. Rothfelder (1998). Improving System Reliability with Automatic Fault Tree Generation. In *Proc. 28th International Symposium on Fault-Tolerant Computing (FTCS'98)*, Munich, Germany, pp. 90–99. IEEE Computer Society Press.

Manian, R., J. Dugan, D. Coppit, and K. Sullivan (1998). Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In *Proc. 3rd International High-Assurance Systems Engineering Symposium (HASE'98)*, pp. 21–28. IEEE Computer Society Press.

Manquinho, V., A. Oliveira, and J. Marques-Silva (1998). Models and Algorithms for Computing Minimum-Size Prime Implicants. In *Proc. International Workshop on Boolean Problems (IWBP'98)*.

Marseguerra, M., E. Zio, J. Devooght, and P. E. Labeau (1998). A concept paper on dynamic reliability via Monte Carlo simulation. *Mathematics and Computers in Simulation 47*, 371–382.

McMillan, K. (1993). *Symbolic Model Checking*. Kluwer Academic Publ.

Papazoglou, I. A. (1994). Markovian Reliability Analysis of Dynamic Systems. In T. Aldemir, N. O. Siu, A. Mosleh, P. C. Cacciabue, and B. G. Göktepe (Eds.), *Reliability and Safety Assessment of Dynamic Process Systems*, Volume 120 of *NATO ASI Series F*, pp. 24–43. Springer-Verlag.

Rae, A. (2000). Automatic Fault Tree Generation - Missile Defence System Case Study. Technical Report 00-36, Software Verification Research Centre, University of Queensland.

Senni, S., M. Semenza, and R. Galvagni (1991). ADMIRA: An Analytical Dynamic Methodology for Integrated Risk Assessment. In G. Apostolakis (Ed.), *Probabilistic Safety Assesment and Management*, pp. 407–412. Elsevier Science.

Siu, N. O. (1994). Risk Assessment for Dynamic Systems: An Overview. *Reliability Engineering ans System Safety 43*, 43–74.

Smidts, C. and J. Devooght (1992). Probabilistic Reactor Dynamics II. A Monte-Carlo Study of a Fast Reactor Transient. *Nuclear Science and Engineering 111*(3), 241–256.

Sullivan, K., J. Dugan, and D. Coppit (1999). The Galileo Fault Tree Analysis Tool. In *Proc. 29th Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*, pp. 232–235. IEEE Computer Society Press.

Vesely, W., F. Goldberg, N. Roberts, and D. Haasl (1981). Fault Tree Handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission.