Symbolic Fault Tree Analysis for Reactive Systems

Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo*

ITC-IRST, Via Sommarive 18, 38050 Trento, Italy
bozzano@irst.itc.it
ph.: +39 0461 314367, fax: +39 0461 302040

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. Generating fault trees is particularly critical in the case of reactive systems, as hazards can be the result of complex interactions involving the dynamics of the system and of the faults. Recently, there has been a growing interest in model-based fault tree analysis using formal methods, and in particular symbolic model checking techniques. In this paper we present a broad range of algorithmic strategies for efficient fault tree analysis, based on binary decision diagrams (BDDs). We describe different algorithms encompassing different directions (forward or backward) for reachability analysis, using dynamic cone of influence techniques to optimize the use of the finite state machine of the system, and dynamically pruning of the frontier states. We evaluate the relative performance of the different algorithms on a set of industrial-size test cases.

1 Introduction

The goal of safety analysis is to investigate the behavior of a system in degraded conditions, that is, when some parts of the system are not working properly, due to malfunctions. Safety analysis includes a set of activities, that have the goal of identifying and characterizing all possible hazards, and are performed in order to ensure that the system meets the safety requirements that are required for its deployment and use. Safety analysis activities are particularly critical in the case of reactive systems, because hazards can be the result of complex interactions involving the dynamics of the system [29].

Recently, there has been a growing interest in model-based safety analysis [8, 6, 4, 1, 15, 25, 17, 18, 9, 7] using formal methods, and in particular symbolic model checking techniques. Traditionally, safety analysis activities are performed manually, and rely on the skill of the safety engineers, hence they are an error-prone and time-consuming activity, and they may rapidly become impractical in case of large and complex systems. Safety analysis based on formal methods, on the other hand, aims at reducing the effort involved and increase the quality of the results, by focusing the effort on building formal models of the system [8, 7], rather than carrying out the analyses.

Fault Tree Analysis (FTA) [35] is one of the most popular safety analysis activities. It is a deductive, top-down method to analyze system design and robustness. It

* This work has been partly supported by the E.U.-sponsored project ISAAC, contract no. AST3-CT-2003-501848.

involves specifying a *top level event* (TLE hereafter) and a set of possible *basic events* (e.g., component faults); the goal is the identification of all possible *cut sets*, i.e. sets of basic events which cause the TLE to occur. Fault trees provide a convenient symbolic representation of the combination of events causing the top level event, and they are usually represented as a parallel or sequential combination of logical gates. To allow a quantitative evaluation, the probability of the events is also included in the fault tree.

In this paper, we focus on the problem of FTA for reactive systems, i.e. systems with infinite behavior over time. The problem is substantially harder than the traditional case, where the system is abstracted and modeled to be state-less and combinational, due to the presence of dynamics, that can influence the presence and the effect of failures.

The main contribution of the paper is the definition, implementation and evaluation of a broad range of algorithms and strategies for efficient fault tree analysis of reactive systems. The algorithms are based on (extended) reachability analysis of a model of the system, and apply to a general framework, where different dynamics for failure mode variables (e.g. persistent and sporadic faults) are possible, thus extending the work in [7]. We point out several distinguishing features. First, fault trees can be constructed both by forward and backward search; it is indeed well known that depending on the structure of the state space of the reactive system, dramatic differences in performance can result, depending on the search direction. Second, backward search is optimized with the use of *dynamic cone of influence* (DCOI), which consists in a lazy generation of restricted models to be used for computing the backward search steps. Third, we propose an optimization called *dynamic pruning*, applicable both to the forward and the backward reachability algorithms, that detects minimal cut sets during the search, and thus can limit the exploration and reduce the number of required iterations.

The algorithms leverage techniques borrowed from symbolic model checking, in particular Binary Decision Diagrams (BDDs for short), that enable the exploration of very large state spaces. The algorithms have been implemented and integrated within FSAP [16, 7], a platform aiming at supporting design and safety engineers in the development and in the safety assessment of complex, reactive systems. The platform automates the generation of artifacts that are typical of reliability analysis, for example failure mode and effect analysis tables, and fault trees. FSAP consists of a graphical interface and an engine based on the NuSMV model checker [23, 11]. NuSMV provides support for user-guided or random simulation, as well as standard model checking capabilities like property verification and counterexample trace generation.

We experimentally evaluate the different algorithms on a set of scalable benchmarks derived from industrial designs. The results show that the forward and backward search directions are indeed complementary, and the proposed optimizations result in a substantial performance improvement.

The paper is structured as follows. In Sect. 2 we give some background about model checking reactive systems; in Sect. 3 we discuss fault tree analysis; in Sect. 4 we describe the algorithms for fault tree generation; in Sect. 5 we outline the implementation in the FSAP platform; in Sect. 6 we present the experimental evaluation; finally, in Sect. 7 we discuss some related work, and in Sect. 8 we draw some conclusions and outline future work.

2 Background

2.1 Modeling Reactive Systems

We are interested in the analysis of reactive systems, whose behavior is potentially infinite over time, such as operating systems, physical plants, and controllers. Reactive systems are modeled as Kripke structures.

Definition 1 (Kripke Structure). *Let $\mathcal{P}$ be a set of propositions. A Kripke structure is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ where: $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states; $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation; $\mathcal{L} : \mathcal{S} \longrightarrow 2^{\mathcal{P}}$ is the labeling function.*

We require the transition relation to be total, i.e. for each state s there exists a successor state s' such that $\mathcal{R}(s, s')$. We notice that Kripke structures can model nondeterministic behavior, by allowing states to have multiple successors. The labeling function associates each state with information on which propositions hold in it.

An execution of the system is modeled as a trace (also called a behavior) in such a Kripke structure, obtained starting from a state $s \in \mathcal{I}$, and then repeatedly appending states reachable through $\mathcal{R}$.

Definition 2 (Trace). *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a Kripke structure. A trace for $\mathcal{M}$ is a sequence $\pi = s_0, s_1, \ldots, s_k$ such that $s_0 \in \mathcal{I}$ and $\mathcal{R}(s_{i-1}, s_i)$ for $1 \leq i \leq k$.*

A state is reachable if and only if there exists a trace containing it. Given the totality of $\mathcal{R}$, a trace can always be extended to have infinite length. We notice that systems are often presented using module composition; the resulting structure can be obtained by composing the sub-structures, but its state space may be exponential in the number of composed modules. In the following we confuse a system and the Kripke structure modeling it; we also assume that a Kripke structure $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is given.

2.2 Symbolic Model Checking

Model checking is a widely used formal verification technique. While testing and simulation may only verify a limited portion of the possible system behaviors, model checking provides a formal guarantee that some given specification is obeyed, i.e. all the traces of the system are within the acceptable traces of the specification. Given a system $\mathcal{M}$, represented as a Kripke structure, and a requirement ϕ, typically specified as a formula in some temporal logic, model checking analyzes whether $\mathcal{M}$ satisfies ϕ, written $\mathcal{M} \models \phi$. Model checking consists in exhaustively exploring every possible system behavior, to check automatically that the specifications are satisfied.

In its simpler form, referred to as *explicit state*, model checking is based on the expansion and storage of individual states. These techniques suffer from the so-called state explosion problem, i.e. they need to explore and store the states of the state transition graph. A major breakthrough was enabled by the introduction of *symbolic model checking* [21]. The idea is to manipulate *sets of* states and transitions, using a logical formalism to represent the characteristic functions of such sets. Since a small logical formula may admit a large number of models, this results in many practical cases in a

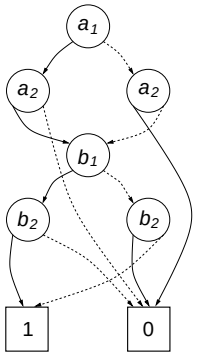

Fig. 1. A BDD for the formula $(a_1 \leftrightarrow a_2) \wedge (b_1 \leftrightarrow b_2)$

very compact representation which can be effectively manipulated. Another key issue is the use of an efficient machinery to carry out the manipulation. For this, we use Ordered Binary Decision Diagrams [10] (BDDs for short).

BDDs are a representation for boolean formulae, which is canonical once an order on the variables has been established. Fig. 1 depicts the BDD for the boolean formula $(a_1 \leftrightarrow a_2) \wedge (b_1 \leftrightarrow b_2)$, using the variable ordering a_1, a_2, b_1, b_2. Solid lines represent "then" arcs (the corresponding variable has to be considered positive), dashed lines represent "else" arcs (the corresponding variable has to be considered negative). Paths from the root to the node labeled with "1" represent the satisfying assignments of the represented boolean formula (e.g., $a_1 \leftarrow 1, a_2 \leftarrow 1, b_1 \leftarrow 0, b_2 \leftarrow 0$). Efficient BDD packages are available. They rely on caching and uniqueness in order to maximize reuse among BDDs, provide memoization for the operations, implement constant time negation by means of pointer's complement, and include a number of efficient re-ordering methods. Despite the worst-case complexity (e.g. certain classes of boolean functions are proved not to have a polynomial-size BDD representation for any order), in practice it is possible to represent Kripke structures effectively.

We now show how a Kripke structure can be symbolically represented. Without loss of generality, we assume that there exists a bijection between $\mathcal{S}$ and $2^{\mathcal{P}}$ (if the cardinality of $\mathcal{S}$ is not a power of two, standard constructions can be applied to extend the Kripke structure). Each state of the system assigns a valuation to each variable in $\mathcal{P}$. We say that a proposition p holds in s, written $s \models p$, if and only if $p \in \mathcal{L}(s)$. Similarly, it is possible to evaluate a boolean formula $\phi(\mathcal{P})$, written $s \models \phi(\mathcal{P})$. The representation of a Kripke structure with BDDs is as follows. For each proposition in $\mathcal{P}$ we introduce a BDD variable; we use $\underline{x}$ to denote the vector of such variables, that we call state variables, and we assume a fixed correspondence between the propositions in $\mathcal{P}$ and the variables in $\underline{x}$. We use $\underline{x}$ to denote the vector of variables representing the states of a given system. We write $\mathcal{I}(\underline{x})$ for the BDD (corresponding to the formula) representing the initial states. To represent the transitions, we introduce a set of "next" variables $\underline{x}'$, used for the state resulting after the transition. A transition from s to s' is then represented as a truth assignment to the current and next variables. We use $\mathcal{R}(\underline{x}, \underline{x}')$ for the formula representing the transition relation expressed in terms of those variables.

Operations over sets of states can be represented by means of boolean operators. For instance, intersection amounts to conjunction between the formulae representing the sets, union is represented by disjunction, complement is represented by negation, and projection is realized with quantification. BDDs provide primitives to compute efficiently all these operations.

3 Fault Tree Analysis for Reactive Systems

Safety analysis is a fundamental step in the design of complex, critical systems. The idea is to analyze the behavior of the system in presence of *faults*, under the hypothesis that components may break down. Model-based safety analysis is carried out on formally specified models which take into account system behavior in the presence of faults. The first step is to identify a set of state variables, called *failure mode variables*, to denote the possible failures, and to identify a set of properties of interest. Intuitively, a failure mode variable is true in a state when the corresponding fault occurs (different failure mode variables are associated to different faults). Once this is done, different forms of analysis investigate the relationship between failures and the occurrence of specific events (called top level events), typically the violation of some of the properties. In the rest of this section, we assume that a set of failure mode variables $\mathcal{F} \subseteq \mathcal{P}$ is given.

Two traditional safety analysis activities are Failure Mode and Effects Analysis (FMEA), and Fault Tree Analysis (FTA). FMEA analyzes which properties are lost, under a specific failure mode configuration, i.e. a truth assignment to the variables in $\mathcal{F}$; the results of the analysis are summarized in a FMEA table. FTA progresses in the opposite direction: given a property, the set of causes has to be identified that can cause the property to be lost. In the rest of this paper we focus on FTA (although many of the techniques described here may be applied also to FMEA).

Traditionally, safety analysis is carried out on a combinational view of the system. Here we specifically focus on reactive systems. In this context, different models of failure may have different impact on the results. Moreover, the temporal relationships between failures may be important, e.g. fault f_1 may be required to occur before another fault f_2 (see [9] for a proposal to enrich the notion of minimal cut set).

Our framework is completely general: it encompasses both the case of *permanent* failure modes (*once failed, always failed*), and the case of *sporadic* or *transient* ones, that is, when faults are allowed to occur sporadically (e.g., a sensor showing an invalid reading for a limited period of time), or when repairing is possible.

Fault tree analysis [35] is an example of deductive analysis, which, given the specification of an undesired condition, usually referred to as a *top level event* (TLE), systematically builds all possible chains of one of more basic faults that contribute to the occurrence of the event. The result of the analysis is a *fault tree*, representing the logical interrelationships of the basic events that lead to the undesired state. In its simpler form [7] a fault tree can be represented with a two-layer logical structure, namely a top level disjunction of the combinations of basic faults causing the top level event. Each combination, which is called *cut set*, is in turn the conjunction of the corresponding basic faults. In general, logical structures with multiple layers can be used, for instance based on the hierarchy of the system model [3]. A cut set is formally defined as follows.

Definition 3 (Cut set). *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a system with a set of failure mode variables $\mathcal{F} \subseteq \mathcal{P}$, let $FC \subseteq \mathcal{F}$ be a fault configuration, and $TLE \in \mathcal{P}$. We say that FC is a cut set of TLE, written $cs(FC, TLE)$ if there exists a trace $s_0, s_1, \ldots, s_k$ for $\mathcal{M}$ such that: $i)$ $s_k \models TLE$; $ii)$ $\forall f \in \mathcal{F} \quad f \in FC \iff \exists i \in \{0, \ldots, k\}\ (s_i \models f)$.*

Intuitively, a cut set corresponds to the set of failure mode variables that are active at some point along a trace witnessing the occurrence of the top level event. Typically, one is interested in isolating the fault configurations that are minimal in terms of failure mode variables, that is, those that represent simpler explanations, in terms of faults, for the occurrence of the top level event. Under the hypothesis of independent faults, a minimal configuration is a more probable explanation with respect to a configuration being a proper superset, hence it has a higher importance in reliability analysis. Minimal configurations are called *minimal cut sets* and are formally defined as follows.

Definition 4 (Minimal Cut Sets). *Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be a system with a set of failure mode variables $\mathcal{F} \subseteq \mathcal{P}$, let $F = 2^{\mathcal{F}}$ be the set of all fault configurations, and $TLE \in \mathcal{P}$. We define the set of cut sets and minimal cut sets of TLE as follows:*

$$\begin{aligned} CS(TLE) &= \{FC \in F \mid cs(FC, TLE)\} \\ MCS(TLE) &= \{cs \in CS(TLE) \mid \forall cs' \in CS(TLE)\ (cs' \subseteq cs \Rightarrow cs' = cs)\} \end{aligned}$$

We remark that the notion of minimal cut set can be extended to the more general notion of *prime implicant* (see [14]), which is based on a different definition of minimality, involving both the activation and the absence of faults. We omit the formal definition for lack of space. We also remark that, although not illustrated in this paper, the fault tree can be complemented with probabilistic information, as done, e.g., in the FSAP platform [16]. The probability of occurrence of the top level event can be computed on the basis of the probabilities of the basic faults.

Based on the previous definitions, fault tree analysis can be described as the activity that, given a TLE, involves the computation of all the minimal cut sets (or prime implicants) and their arrangement in the form of a tree.

4 Symbolic Fault Tree Analysis

We now review the different algorithms for fault tree generation, available in FSAP. We start in Sect. 4.1 with the standard, forward algorithm that we described in [7]. In this context, we extend the algorithm in order to support sporadic failure modes, which were not allowed in [7]. Then, we describe a novel backward algorithm in Sect. 4.2, and its optimization based on dynamic cone of influence in Sect. 4.3; finally, in Sect. 4.4 we introduce an optimization called *dynamic pruning*, which is applicable both to the forward and the backward algorithms.

In the following, we assume that a Kripke structure $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ is given, and we use the following notations. First, $\underline{f}$ denotes the vector of failure variables. Given two vectors $\underline{v} = v_1 \ldots v_k$ and $\underline{w} = w_1 \ldots w_k$, the notation $\underline{v} = \underline{w}$ stands for $\bigwedge_{i=1}^{k}(v_i = w_i)$. We use $ITE(p, q, r)$ (if-then-else) to denote $((p \rightarrow q) \wedge (\neg p \rightarrow r))$. Finally, given a set of states Q, the image of Q is defined as $\{s' \mid \exists s \in Q.\ \mathcal{R}(s, s')\}$. It can

function FTA-Forward ($\mathcal{M}, Tle$)
1 $\mathcal{M} := Extend(\mathcal{M}, \mathcal{R}^o)$;
2 $Reach := \mathcal{I} \cap (\underline{o} = \underline{f})$;
3 $Front := \mathcal{I} \cap (\underline{o} = \underline{f})$;
4 **while** ($Front \neq \emptyset$) **do**
5 $temp := Reach$;
6 $Reach := Reach \cup fwd\_img(\mathcal{M}, Front)$;
7 $Front := Reach \backslash temp$;
8 **end while**;
9 $CS := Proj(\underline{o}, Reach \cap Tle)$;
10 $MCS := Minimize(CS)$;
11 **return** $Map_{\underline{o} \rightarrow \underline{f}}(MCS)$;

function FTA-Backward ($\mathcal{M}, Tle$)
1 $\mathcal{M} := Extend(\mathcal{M}, \mathcal{R}^g)$;
2 $Reach := Tle \cap (\underline{g} = \underline{f})$;
3 $Front := Tle \cap (\underline{g} = \underline{f})$;
4 **while** ($Front \neq \emptyset$) **do**
5 $temp := Reach$;
6 $Reach := Reach \cup bwd\_img(\mathcal{M}, Front)$;
7 $Front := Reach \backslash temp$;
8 **end while**;
9 $CS := Proj(\underline{g}, Reach \cap \mathcal{I})$;
10 $MCS := Minimize(CS)$;
11 **return** $Map_{\underline{g} \rightarrow \underline{f}}(MCS)$;

Fig. 2. Forward and backward algorithms

be encoded symbolically as follows: $fwd\_img(\mathcal{M}, Q(\underline{x})) = \exists \underline{x}.\ (Q(\underline{x}) \wedge \mathcal{R}(\underline{x}, \underline{x}'))$. Similarly, the preimage of Q is defined as $\{s \mid \exists s' \in Q.\ \mathcal{R}(s, s')\}$ and can be encoded as $bwd\_img(\mathcal{M}, Q(\underline{x}')) = \exists \underline{x}'.\ (Q(\underline{x}') \wedge \mathcal{R}(\underline{x}, \underline{x}'))$.

4.1 Forward Fault Tree Analysis

The forward algorithm starts from the initial states of the system and accumulates, at each iteration, the forward image. In order to take into account sporadic failure modes (compare Def. 3, condition ii), at each iteration we need to "remember" if a failure mode has been activated. To this aim, for each failure mode variable $f_i \in \mathcal{F}$, we introduce an additional variable o_i (*once* f_i), which is true if and only if f_i has been true at some point in the past. This construction is traditionally referred to as *history variable*, and is formalized by the transition relation $\mathcal{R}^o$ given by the following condition:

$$\bigwedge_{f_i \in \mathcal{F}} ITE(o_i, o_i', o_i' \leftrightarrow f_i')$$

Let $Extend(\mathcal{M}, \mathcal{R}^o)$ be the Kripke structure obtained from $\mathcal{M}$ by replacing the transition relation $\mathcal{R}$ with the synchronous product between $\mathcal{R}$ and $\mathcal{R}^o$, in symbols $\mathcal{R}(\underline{x}, \underline{x}') \wedge \mathcal{R}^o(\underline{x}, \underline{x}')$ and modifying the labeling function $\mathcal{L}$ accordingly.

The pseudo-code of the algorithm is described in Fig. 2 (left). The inputs are $\mathcal{M}$ and Tle (the set of states satisfying the top level event). A variable $Reach$ is used to accumulate the reachable states, and a variable $Front$ to keep the *frontier*, i.e. the newly generated states (at each step, the image operator needs to be applied only to the latter set). Both variables are initialized with the initial states, and the history variables with the same value as the corresponding failure mode variables. The core of the algorithm (lines 4-8) computes the set of reachable states by applying the image operator to the frontier until a fixpoint is reached (i.e, the frontier is the empty set). The resulting set is intersected (line 9) with Tle, and projected over the history variables. Finally, the

minimal cut sets are computed (line 10) and the result is mapped back from the history variables to the corresponding failure mode variables (line 11).

Note that all the primitives used in algorithm, including the minimization routine (see [14, 26]) can be realized using BDD data structures, as explained in Sect. 2.2. For instance, set difference (line 7) can be defined as $Reach(\underline{x}) \wedge \neg temp(\underline{x})$, and the mapping function (line 11) as $Map_{\underline{o} \rightarrow \underline{f}}(\phi(\underline{o})) = \exists \underline{o}.(\phi(\underline{o}) \wedge (\underline{o} = \underline{f}))$.

4.2 Backward Fault Tree Analysis

The backward algorithm performs reachability via the preimage operator bwd_img. This time, we need to "remember" if a failure mode has been activated at some step *in the future*. To this aim, for each $f_i \in \mathcal{F}$ we introduce an additional variable g_i (these variables are referred to as *guess* or *prophecy variables*, and they can be seen as the dual of history variables). Let $\mathcal{R}^g$ be the transition relation defined by the condition $\bigwedge_{f_i \in \mathcal{F}} ITE(g_i', g_i, g_i \leftrightarrow f_i)$ and $Extend(\mathcal{M}, \mathcal{R}^g)$ be defined as in Sect. 4.1.

The backward algorithm is presented in Fig. 2 (right). The core (lines 4-8) is similar to forward one. It starts from the set of states satisfying the top level event, with the additional constraints that the prophecy variables have been initialized with the same value as the corresponding failure mode variables, and it performs backward reachability until a fixpoint is reached. The resulting set is intersected (line 9) with the initial states, and projected over the prophecy variables. Finally, the minimal cut sets are computed (line 10) and the result is mapped back to the failure mode variables (line 11).

4.3 Backward Fault Tree Analysis With Dynamic Cone Of Influence

The algorithm for backward fault tree analysis can be optimized by means of the following technique, referred to as Dynamic Cone of Influence reduction (DCOI). The idea is based on the fact that often models enjoy some local structure, so that the next value of a certain variable only depends on a subset of the whole state variables. Suppose that the top level event Tle only depends on a limited set of variables, say $\underline{x}^0 \subset \underline{x}$. Thus, when computing the preimage of Tle (line 6 in Fig. 2, right), it is possible to consider only those parts of the Kripke structure that influence the next value of Tle. Such a restricted Kripke structure, referred to as $\mathcal{M}^0$, is typically much simpler than the whole $\mathcal{M}$, and preimages can be computed much more effectively. The resulting preimage may also depend on a restricted set of variables $\underline{x}^1 \subset \underline{x}$, and at the second step the corresponding $\mathcal{M}^1$ is used to compute the preimage. The process is iterated until a fix point is reached; in the worst case, the whole machine is taken into account, but it is possible that convergence is reached before the whole $\mathcal{M}$ is constructed.

The structure of the algorithm (see Fig. 3, left) is the same as the standard backward algorithm. However, at each step i, a different Kripke structure is used, instead of the global $\mathcal{M}$ (lines 7 and 8); the $dcoi\_get$ primitive encapsulates the process of lazily constructing the Kripke structure necessary for the i-th preimage computation.

4.4 Dynamic Pruning

All the algorithms previously discussed can be optimized by using dynamic pruning. We describe below the extension of the forward algorithm (the other ones can be extended

```
function FTA-Backward-DCOI (M, Tle)
1   i := 0;
2   M := Extend(M, R^g);
3   Reach := Tle ∩ (g = f);
4   Front := Tle ∩ (g = f);
5   while (Front ≠ ∅) do
6     temp := Reach;
7     M^i = dcoi_get(M, Tle, i);
8     Reach := Reach ∪
                bwd_img(M^i, Front);
9     Front := Reach\temp;
10    i := i + 1;
11  end while;
12  CS := Proj(g, Reach ∩ I);
13  MCS := Minimize(CS);
14  return Map_{g→f}(MCS);
```

```
function FTA-Forward-Pruning (M, Tle)
1   CS := ∅;
2   M := Extend(M, R^o);
3   Reach := I ∩ (o = f);
4   Front := I ∩ (o = f);
5   while (Front ≠ ∅) do
6     CS := CS ∪ Proj(o, Reach ∩ Tle);
7     temp := Reach;
8     Reach := Reach ∪
                fwd_img(M, Front);
9     Front := Reach\temp;
10    Front := Front\Widen(CS);
11  end while;
12  MCS := Minimize(CS);
13  return Map_{o→f}(MCS);
```

Fig. 3. Backward algorithm, using DCOI; forward algorithm, with pruning

similarly). The idea is that, at each iteration (lines 4-8 in Fig. 2), it is safe to discard a state, provided we know that it will not contribute to the set of *minimal* cut sets. In particular, this is true whenever we know that the failure modes being active in that state are a superset of a fault configuration that has already been proved to be a cut set. The implementation (see Fig. 3, right) is as follows. At each iteration (line 6) a partial set of cut sets CS is computed. Based on this set, all the states on the frontier, whose active failure modes are a superset of any fault configuration in CS, are pruned (line 10). The primitive $Widen$ is defined as $Widen(CS) = \{s \mid \exists cs \in CS\ (cs \subseteq s)\}$. Intuitively, it collects all the states that include any element in CS as a proper subset (the definition can be extended to the more general case of *prime implicants*). Typically, the use of dynamic pruning may result in a significant reduction of the search space.

5 Implementation in the FSAP Platform

In this section we discuss the implementation of the algorithms described in Sect. 4 in the FSAP platform [16, 7]. As advocated in [8], it is important to have a complete decoupling between the system model and the fault model. For this reason, the FSAP platform relies on the notions of *nominal system model* and *extended system model*. The nominal model formalizes the behavior of the system when it is working as expected, whereas the extended model defines the behavior of the system in presence of faults.

The decoupling between the two models is achieved in the FSAP platform by generating the extended model automatically via a so-called *model extension step*. Model extension takes as input a system and a specification of the faults to be added, and automatically generates the corresponding extended system. It can be formalized as follows. Let $\mathcal{M} = \langle \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} \rangle$ be the nominal system model. A fault is defined by the propo-

sition $p \in \mathcal{P}$ to which it must be attached to, and by its type, specifying the "faulty behavior" of p in the extended system (e.g., p can non-deterministically assume a random value, or be stuck at a particular value). FSAP introduces a new proposition p^{FM}, the *failure mode variable*, modeling the possible occurrence of the fault, and two further propositions p^{Failed} and p^{Ext}, with the following intuitive meaning. The proposition p^{Failed} models the behavior of p when a fault has occurred. For instance, the following condition (where $\mathcal{S}'$ is the set of states of the extended system) defines a so-called *inverted* failure mode (that is, p^{Failed} holds if and only if p does not hold):

$$\forall s \in \mathcal{S}' \quad (s \models p^{Failed} \iff s \not\models p) \tag{1}$$

The proposition p^{Ext} models the extended behavior of p, that is, it behaves as the original p when no fault is active, whereas it behaves as p^{Failed} in presence of a fault:

$$\forall s \in \mathcal{S}' \quad s \not\models p^{FM} \Rightarrow (s \models p^{Ext} \iff s \models p) \tag{2}$$

$$\forall s \in \mathcal{S}' \quad s \models p^{FM} \Rightarrow (s \models p^{Ext} \iff s \models p^{Failed}) \tag{3}$$

The extended system $\mathcal{M}^{Ext} = \langle \mathcal{S}', \mathcal{I}', \mathcal{R}', \mathcal{L}' \rangle$ can be easily defined in terms of the nominal system by adding the new propositions, modifying the definition of the (initial) states and of the transition relation, and imposing the conditions (1) (for an inverted failure mode), (2) and (3). We omit the details for the sake of simplicity. Finally, system extension with respect to a *set* of propositions can be defined in a straightforward manner, by iterating system extension over single propositions.

The system model resulting from the extension step is used in FSAP to carry out the analyses. The algorithms described in Sect. 4 are implemented in FSAP on top of the NuSMV tool [23, 11], using BDDs as explained in Sect. 2.2 (we refer to Sect. 8 for a discussion on alternative algorithms). The FSAP platform can be used to compute both the minimal cut sets and the prime implicants of a given top level event. In addition, FSAP can compute the probability of the top level event, on the basis of the probabilities of the basic faults, under the hypothesis of independent faults.

6 Experimental Evaluation

In this section we describe the experimental evaluation we carried out. Five algorithms have been evaluated: forward algorithm (FWD in the following), forward algorithm with dynamic pruning (FWD-PRUN), backward algorithm (BWD), backward algorithm with DCOI (DCOI), and backward algorithm with DCOI and dynamic pruning (DCOI-PRUN). Two different test-cases have been used. Both models are of industrial size and have been developed inside the ISAAC project[1]. We remark that the models are covered by a non-disclosure agreement, hence they are only briefly described.

The first model (referred to as "TDS model") is a model of a subsystem of an aircraft. It consists of a mechanical and a pneumatic line, driving a set of utilities, and being controlled by a central unit. Faults are attached to different components of the two lines. We ran different experiments by limiting the faults that can be active at any

[1] `http://www.isaac-fp6.org`

time: in the simplest experiment only 2 faults out of 34 are active. The second test-case (referred to as "Cassini model") is a model of the propulsion system of the Cassini-Huygens spacecraft. The propulsion system is composed of two engines fed by redundant propellant/gas circuit lines. Each line contains several valves and pyrovalves (a pyrovalve being a pyrotechnically actuated valve, whose status – open or close– can be changed at most once). Faults are attached to the engines, the propellant/gas tanks, and the (pyro)valves. We built several variants of the Cassini model by modifying (increasing the redundancy of) the (pyro)valve layer located between the propellant tanks and the engines. The property used to generate the fault tree is related to a correct input pressure in (at least one of the) engines in presence of a correct output pressure from the gas and the propellant tanks.

We ran each experiment with a different invocation of the model checker. For each model, we list the number of minimal cut sets (column **MCS**). For each run, we report the total usage of time (column **T**, in seconds – in parentheses the fraction of time spent for compiling the model) and memory (column **M**, in Mb), and the number of iterations needed to reach the fixpoint (column **K**). Compilation includes parsing, encoding of the variables, and – for all the algorithms except DCOI and DCOI-PRUN – building of the Kripke structure into BDD (the low compilation time for DCOI and DCOI-PRUN is due to the fact that building of the Kripke structure is delayed). The experiments have been run on a 2-processor machine equipped with Intel Xeon 3.00GHz, with 4Gb of memory, running Linux RedHat Enterprise 4 (only one processor was allowed to run for each experiment). The time limit was set to 1 hour and the memory limit to 1 Gb. A '↑' in the time or memory columns stands for a time-out or memory-out, respectively.

The algorithms have been implemented in NuSMV 2.4.1, and run with the following options. In both cases, we used static variable ordering. The motivation is to allow for a fair comparison between the relative performances of the different algorithms, as dynamic re-ordering could affect the performances in an unpredictable manner, depending on how the re-ordering is performed inside the BDD package. We notice that, in general, a good variable ordering is crucial to achieve the best performances, and typically dynamic re-ordering over-performs static ordering. For the TDS model, given the high complexity of the model, we used an off-line pre-computed variable ordering, which has been passed to NuSMV at command line (option `-i`). For the Cassini model, we used a pre-defined static ordering available in NuSMV (option `-vars_order lexicographic`). Furthermore, for the Cassini model we disabled conjunctive partitioning (option `-mono`), in order to purify the results of the DCOI and DCOI-PRUN algorithms from the effect of a better of worse partitioning choice (for the TDS model, this option was not necessary, as it will be evident from the experimental data).

The experimental results are reported in Figs. 4 and 5. The first comment is that there is great variance of performance in the different algorithms. Proceeding forwards appears to be extremely effective in the TDS model, while for the Cassini model, backward search is very effective. This can be partly justified by the different structure of the two models: complex but mostly flat for the TDS model, deeply layered (with the level of layering increasing with the complexity of the model instance) for the Cassini model. In general, as in model checking, the nature of the state spaces may dramatically

NR_FAIL	MCS	FWD			FWD-PRUN			BWD			DCOI			DCOI-PRUN		
		T	M	K	T	M	K	T	M	K	T	M	K	T	M	K
2	2	58.7 (8.2)	30	62	9.3 (8.2)	26	11	↑	-	-	↑	-	-	↑	-	-
3	3	102.9 (8.5)	47	67	10.0 (8.5)	26	14	↑	-	-	↑	-	-	↑	-	-
4	4	259.0 (8.6)	67	69	12.1 (8.6)	27	14	↑	-	-	↑	-	-	↑	-	-
5	5	616.3 (8.9)	138	69	15.7 (8.9)	26	14	↑	-	-	↑	-	-	↑	-	-
6	6	521.6 (9.4)	96	69	13.8 (9.4)	27	14	↑	-	-	↑	-	-	↑	-	-
7	7	609.0 (9.5)	131	69	16.2 (9.5)	27	14	↑	-	-	↑	-	-	↑	-	-
8	8	656.3 (9.6)	124	69	16.5 (9.6)	27	14	↑	-	-	↑	-	-	↑	-	-
9	8	1141.9 (10.0)	187	69	16.7 (10.0)	27	14	↑	-	-	↑	-	-	↑	-	-
10	8	2371.5 (10.1)	318	69	19.7 (10.1)	27	14	↑	-	-	↑	-	-	↑	-	-
11	8	↑	-	-	16.2 (10.2)	27	14	↑	-	-	↑	-	-	↑	-	-
12	8	↑	-	-	17.4 (10.4)	27	14	↑	-	-	↑	-	-	↑	-	-
13	8	↑	-	-	20.9 (10.7)	28	14	↑	-	-	↑	-	-	↑	-	-
14	8	↑	-	-	32.4 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
15	8	↑	-	-	25.2 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
16	8	↑	-	-	26.3 (11.0)	28	14	↑	-	-	↑	-	-	↑	-	-
17	8	↑	-	-	25.0 (11.7)	28	14	↑	-	-	↑	-	-	↑	-	-
18	8	↑	-	-	26.2 (12.0)	29	14	↑	-	-	↑	-	-	↑	-	-
19	8	↑	-	-	27.1 (12.3)	29	14	↑	-	-	↑	-	-	↑	-	-
20	8	↑	-	-	30.5 (12.5)	29	14	↑	-	-	↑	-	-	↑	-	-
34	12	↑	-	-	1219.0 (17.8)	129	14	↑	-	-	↑	-	-	↑	-	-

Fig. 4. Experimental results for the TDS model

MODEL	MCS	FWD			FWD-PRUN			BWD			DCOI			DCOI-PRUN		
		T	M	K	T	M	K	T	M	K	T	M	K	T	M	K
v-2222	329	6.7 (3.2)	15	5	9.0 (3.2)	16	5	37.6 (3.2)	14	5	1.5 (0.1)	13	4	1.5 (0.1)	13	4
v-3222	401	29.0 (3.6)	19	6	14.0 (3.6)	19	6	↑	-	-	1.8 (0.1)	14	5	1.8 (0.1)	15	5
v-3322	489	↑	-	-	29.8 (4.4)	26	6	↑	-	-	2.6 (0.1)	17	5	2.7 (0.1)	17	5
v-3332	599	↑	-	-	578.3 (4.7)	30	6	↑	-	-	3.4 (0.1)	17	5	3.5 (0.1)	17	5
v-3333	734	↑	-	-	↑	-	-	↑	-	-	4.2 (0.1)	17	5	4.3 (0.1)	17	5
v-4333	869	↑	-	-	↑	-	-	↑	-	-	5.5 (0.1)	18	6	6.1 (0.1)	18	6
v-4433	1029	↑	-	-	↑	-	-	↑	-	-	7.3 (0.1)	18	6	7.4 (0.1)	18	6
v-4443	1221	↑	-	-	↑	-	-	↑	-	-	9.4 (0.2)	19	6	10.1 (0.2)	19	6
v-4444	1449	↑	-	-	↑	-	-	↑	-	-	11.9 (0.2)	19	6	12.6 (0.2)	19	6
v-5444	1667	↑	-	-	↑	-	-	↑	-	-	17.8 (0.2)	20	7	17.7 (0.2)	20	7
v-5544	1941	↑	-	-	↑	-	-	↑	-	-	25.0 (0.2)	21	7	28.3 (0.2)	24	7
v-5554	-	↑	-	-	↑	-	-	↑	-	-	↑	-	-	↑	-	-

Fig. 5. Experimental results for the Cassini model

vary depending on the search direction, and it is not predictable in advance. We claim that, for this reason, it is very important to have available different styles of search.

Second, we notice that the proposed optimizations are always effective, or unnoticeable. DCOI results in dramatic savings for the Cassini model (in fact, the experiments show that DCOI is the crucial factor that makes backward search a winning strategy over forward search). Dynamic pruning has a minor impact in the case of backward search on the Cassini model, but is an enabling factor for TDS, where it substantially reduces the number of iterations needed to reach convergence. In general, dynamic pruning is more effective when lower-order cut sets are found earlier in the search (that is, they are witnessed by shorter traces), hence its effectiveness is highly model-dependent.

7 Related Work

A large amount of work has been done in the area of probabilistic safety assessment (PSA) and in particular on *dynamic reliability* [29]. 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 [20]. For different approaches to dynamic reliability see, e.g., [2, 24, 12, 20, 30]. These approaches are mostly concerned with the evaluation (as opposed to generation) of a given fault tree. Concerning fault tree evaluation, we also mention DIFTree [19], a methodology for the analysis of dynamic fault trees, implemented in the Galileo tool [31]. The methodology is able to identify independent sub-trees, translate them into suitable models, analyze them and integrate the results of the evaluation. Different techniques can be used for the evaluation, e.g., BDD-based techniques, Markov techniques or Monte Carlo simulation. Concerning fault tree validation, we mention [28, 32], both concerned with automatically proving the consistency of fault trees using model checking techniques; [32] presents a fault tree semantics based on Clocked CTL (CCTL) and uses timed automata for system specification, whereas [28] presents a fault tree semantics based on the Duration Calculus with Liveness (DCL) and uses Phase Automata as an operational model.

The FSAP platform has been developed within ESACS[2] (Enhanced Safety Assessment for Complex Systems) and ISAAC[1] (Improvement of Safety Activities on Aeronautical Complex systems), two European-Union-sponsored projects involving various research centers and industries from the avionics sector. For a more detailed description of the project goals we refer to [8, 6, 9]. Within the project, the same methodology has been also implemented in other platforms, see e.g. [4, 1, 15, 25]. Regarding model-based safety analysis, we mention [17, 18], sharing some similarities with the ISAAC approach. In particular, the integration of the traditional development activities with the safety analysis activities, based on a formal model of the system, and the clear separation between the nominal model and the fault model, are ideas that have been pioneered by ESACS [8]. The authors propose to integrate this approach into the traditional "V" safety assessment process. Finally, we mention [22, 33, 34], sharing with ISAAC the application field (i.e., avionics), and the use of NuSMV as a target verification language.

[2] `http://www.esacs.org`

Finally, the routines used to extract the set of minimal cut sets are based on classical procedures for *minimization* of boolean functions, specifically on the implicit-search procedures described in [13, 14, 26, 27], based on Binary Decision Diagrams [10].

8 Conclusions

In this paper we have presented a broad range of algorithmic strategies for efficient fault tree analysis of reactive systems. In particular, we have described algorithms encompassing different directions for reachability analysis, and some useful optimizations. The experimental evaluation showed the complementarity of the search directions and confirmed the impact of the optimizations on the overall performance.

In the future, we intend to investigate the following research directions. First, we intend to explore the automatic combination of forward and backward search. Second, we will explore an alternative symbolic implementation, using SAT-based bounded model checking techniques [5]. As opposed to BDDs, that work by saturating sets of states, these techniques are typically used to find single traces of bounded length. The challenge is to make these techniques complete; a possible solution could be the generalization of induction techniques. Furthermore, we also want to investigate a "hybrid" approach combining BDD-based and SAT-based techniques into the same routine.

Acknowledgments We wish to thank Antonella Cavallo from Alenia Aeronautica and Massimo Cifaldi from AleniaSIA for allowing us to use the TDS model.

References

1. P.A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing Safe, Reliable Systems using Scade. In *Proc. ISoLA 2004*, 2004.
2. T. Aldemir. Computer-assisted Markov Failure Modeling of Process Control Systems. *IEEE Transactions on Reliability*, R-36:133–144, 1987.
3. R. Banach and M. Bozzano. Retrenchment, and the Generation of Fault Trees for Static, Dynamic and Cyclic Systems. In *Proc. SAFECOMP 2006*. Springer, 2006.
4. P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proc. EDCC-4*. Springer, 2002.
5. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS 1999*. Springer, 1999.
6. M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving Safety Assessment of Complex Systems: An industrial case study. In *Proc. FM 2003*. Springer, 2003.
7. M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *Software Tools for Technology Transfer*, 9(1):5–24, 2007.
8. M. Bozzano et al. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *Proc. ESREL 2003*. Balkema Publisher, 2003.
9. M. Bozzano et al. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In *Proc. ERTS 2006*, 2006.
10. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
11. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

12. G. Cojazzi, J. M. Izquierdo, E. Meléndez, and M. S. Perea. 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.*, 1992.
13. O. Coudert and J.C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. DAC 1992*. IEEE Computer Society, 1992.
14. O. Coudert and J.C. Madre. Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In *Proc. RAMS 1993*, 1993.
15. J. Deneux and O. Åkerlund. A Common Framework for Design and Safety Analyses using Formal Methods. In *Proc. PSAM7/ESREL'04*, 2004.
16. The FSAP platform. `http://sra.itc.it/tools/FSAP`.
17. A. Joshi and M.P.E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *Proc. SAFECOMP 2005*. Springer, 2005.
18. A. Joshi, S.P. Miller, M. Whalen, and M.P.E. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proc. DASC 2005*, 2005.
19. R. Manian, J.B. Dugan, D. Coppit, and K.J. Sullivan. Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In *Proc. HASE 1998*. IEEE Computer Society, 1998.
20. M. Marseguerra, E. Zio, J. Devooght, and P. E. Labeau. A concept paper on dynamic reliability via Monte Carlo simulation. *Math. and Comp. in Simulation*, 47:371–382, 1998.
21. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.
22. S.P. Miller, A.C. Tribble, and M.P.E. Heimdahl. Proving the Shalls. In *Proc. FM 2003*. Springer, 2003.
23. The NuSMV model checker. `http://nusmv.itc.it`.
24. I.A. Papazoglou. Markovian Reliability Analysis of Dynamic Systems. In *Reliability and Safety Assessment of Dynamic Process Systems*, pages 24–43. Springer, 1994.
25. T. Peikenkamp, E. Böede, I. Brückner, H. Spenke, M. Bretschneider, and H.-J. Holberg. Model-based Safety Analysis of a Flap Control System. In *Proc. INCOSE 2004*, 2004.
26. A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering and System Safety*, 40(3):203–211, 1993.
27. A. Rauzy and Y. Dutuit. Exact and Truncated Computations of Prime Implicants of Coherent and Non-Coherent Fault Trees within Aralia. *Reliability Engineering and System Safety*, 58(2):127–144, 1997.
28. A. Schäfer. Combining Real-Time Model-Checking and Fault Tree Analysis. In *Proc. FM 2003*. Springer, 2003.
29. N. O. Siu. Risk Assessment for Dynamic Systems: An Overview. *Reliability Engineering and System Safety*, 43:43–74, 1994.
30. C. Smidts and J. Devooght. Probabilistic Reactor Dynamics II. A Monte-Carlo Study of a Fast Reactor Transient. *Nuclear Science and Engineering*, 111(3):241–256, 1992.
31. K.J. Sullivan, J.B. Dugan, and D. Coppit. The Galileo Fault Tree Analysis Tool. In *Proc. FTCS 1999*. IEEE Computer Society, 1999.
32. A. Thums and G. Schellhorn. Model Checking FTA. In *Proc. FM 2003*. Springer, 2003.
33. A.C. Tribble, D.L. Lempia, and S.P. Miller. Software Safety Analysis of a Flight Guidance System. In *Proc. DASC 2002*, 2002.
34. A.C. Tribble and S.P. Miller. Software Safety Analysis of a Flight Management System Vertical Navigation Function - A Status Report. In *Proc. DASC 2003*, 2003.
35. W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. Fault Tree Handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission, 1981.