From Fault-Trees to Safety Conditions

Paolo Torrini, Paul Caspi, Pascal Raymond

Verimag Research Report n^o TR-2007-6

From Fault-Trees to Safety Conditions

Paolo Torrini, Paul Caspi, Pascal Raymond

Verimag-CNRS

Abstract

Fault trees may be used in order to decompose the verification of a safety critical control system into two steps: a high level modelling of the possible faults, from which safety conditions can be automatically extracted; and the model-checking of these conditions with respect to a model of the controller program. To this purpose, we introduce a fault tree formalism with unconventional features related to the modelling of discrete event system; we give this formalism a temporal semantics; and we provide an algorithm essentially related to minimal cut set analysis for the extraction of safety conditions.

Keywords: Fault trees, safety conditions, formal verification

Reviewers: Karine Altisen

Notes: This work has been partially supported by the Predit project PROOFER (2006-2007) whose participants are RATP, Alcatel, Prover Technology and Verimag.

How to cite this report:

@techreport { rr,
title = { From Fault-Trees to Safety Conditions},
authors = { Paolo Torrini, Paul Caspi, Pascal Raymond},
institution = { Verimag Research Report },
number = {TR-2007-6},
year = { 2007},
note = { }
}

Contents

1 Introduction

1.1 The PROOFER project

The work presented in this report was motivated and supported by the Predit project PROOFER. The plan was to apply formal verification methods to prove the safety of a railway control system. The parties involved with the project were

- RATP (running Paris underground railways) as the ordering party,
- Alcatel TSA (now Thales) as system provider,
- Prover-Technology as formal verification tool provider,
- Verimag as academic consultant.

The main goal was to express and prove safety properties on an interlocking control system (e.g. that the controller can prevent derailment). A necessary step was the derivation of controller properties suitable to the verifier from expert knowledge expressed in terms of high level notions, such as that of derailment event. In this report, we are going to present a method based on fault-trees that can be used to extract a controller specification from a high-level representation of failure events.

1.2 Safety conditions in safety-critical systems

Safety critical applications of control systems pose increasingly hard problems of validation and verification. There are several methods that have been applied to problems of significant size — for example, proved construction *à la* B [1], controller synthesis [13], model-checking [14]. However, in practice, many real-world systems are too complex to be handled with such methods. Interlocking systems are no exception.

By railway interlocking systems here we mean programs that can observe railway traffic, control arrangements of signals, and supervise commands to track elements such as switches at junctions and barriers at level crossings, in order to prevent conflicts that may lead to accidents such as derailment and collisions.

In principle, it should be possible to:

- model the environment, *i. e.*, the way trains move on tracks, stop at red lights, the way switches behave under control, etc.;
- model the accidents that the controller should prevent, such as collisions and derailment;
- either formally synthesise a controller, or else build one and model-check that it prevents all the accidents under the assumptions provided by a model of the environment.

However, the size of a system may be too large for this approach to be used with the available tools at a reasonable cost. This appears to have led Prover Technology [16], our partner in the Proofer Project, to adopt a different, two-step approach:

1. expect railway experts to extract from an informal modelling of the system conditions that depend on variables, either observable or controllable by the control program;
2. model-check the control program against those conditions.

In the Proofer Project, our aim is to contribute a specific method for the extraction of safety conditions from high-level models of system failures. In practice, such models are going to be defined under the supervision of railway specialists, hence we find it convenient to rely on a modelling approach which is popular with the industry — and this is the case for the fault tree method.

1.3 The Fault Tree method

The fault tree method is a semi-formal technique used to analyse system failures at the level of design. It has been around for several decades [17], with application to stochastic risk assessment as well as to verification. The method consists of a top-down analysis which starts from prospected failures and looks for its possible causes. It relies on a representation of causal consequence between events which is closely associated with AND-OR trees. It is relatively independent of any low-level modelling of the system [8].

A fault tree is essentially a bipartite graph where two classes of nodes are associated, respectively, to system events and causal relations (called *gates*). Each event may have at most one parent and one child node, so the branching factor depends ultimately on gates. Gates represent causal connections between *input events* (child nodes) and an *output event* (parent node). Top and leaf nodes must be events. Each gate together with its parent and child nodes forms a *component* of the tree. The top node is associated with the main failure under scrutiny. Leaf nodes are associated with *basic events* — usually either system events that do not require further analysis, or conditions such as environment ones.

The type of a gate is defined by the logical character, and sometimes by the temporal and causal one, of the relation associated with it. The logical character consists normally of either a conjunctive or a disjunctive Boolean sub-relation between the inputs, so that the relation may be either of the AND sort — the output event is caused by a conjunction of inputs — or of the OR sort — the output event is caused by any one of the inputs. OR gates may be used to introduce alternative causal explanations. Fault trees in the simplest form are essentially AND-OR trees.

In general, gate types will also depend on temporal sub-relations between the inputs and between inputs and output. When the temporal relation between input and output is one of synchronicity — and thus time can be abstracted away without loss — the gate can be said to be *decompositional*, otherwise *temporal*. A gate may be called *dynamic* when the associated relation involves a constraint about the order in which input events take place, *static* otherwise. Examples of gate types are *logical and* (decompositional), *primary and* (dynamic, inputs taking place in a specified order), *asynchronous and* (static, inputs simply taking place before the output).

In any case, every relation should at least express either an “if” or an “only if” dependency between input events and output — corresponding to what may be called *correctness* and *completeness* aspects of causal analysis, respectively. In general, given our interest in safety, the completeness aspect is ultimately the one that matters — in principle, we need to consider all the possible *necessary* causes. However, the notion of correctness, *i.e.* of what counts as *sufficient* causes, is important for the definition of safety criteria which are as liberal as possible.

1.4 Minimal cut sets

The following notion plays a central role in failure analysis.

Given an AND-OR tree H with origin h and a subset K of the leaves of H, we say that K is a *minimal cut set* of H whenever K is the value for h of a function f from nodes to sets of nodes that satisfies the following specification:
(A) if x is a leaf, then $f\ x\ = \{x\}$;
else, let $y_0, \ldots, y_n$ be the child nodes of x;
(B) if x is an AND node, then $f\ x = \bigcup_{i \leq n}(f\ y_i)$;
(C) if x is an OR node, then $f\ x = f\ y_i$, for any $i \leq n$.

Intuitively, minimal cut sets represent the smallest possible sets of basic events which, under the correctness aspect of the fault tree, suffice to cause the main failure.

1.5 Basic fault tree analysis

The notion of minimal cut set may also be expressed, abstracting completely from time, in a simple logic of conjunction and disjunction, by interpreting nodes as atomic formulas, sets of nodes as conjunctions of formulas, and sets of sets of nodes as disjunctions of formulas. The minimal cut sets of a fault tree can then be represented, according to the definition given above, as a formula in disjunctive normal form, i. e.

$$(x_{11} \vee \ldots \vee x_{1m}) \wedge \ldots \wedge (x_{j1} \vee \ldots \vee x_{jn})$$

where all atomic formulas represent leaf nodes.

Fault Tree Analysis (*FTA*) at its simplest, abstracting from time relations, consists of the computation of minimal cut sets over the AND-OR structure of fault trees, returning a Boolean formula in disjunctive normal form.

It is a known fact — see [3], for example — that minimal cut sets (without time) can be computed by inference in a deductive system as simple as a logic with implication, conjunction and disjunction — this is the case for Boolean logic, as well as for weaker logics such as positive and intuitionistic ones.

Allowing for abstraction from time, as we are doing now, the truth of x may be taken as the statement "x happens", $\wedge, \vee, \neg$ may be given the standard intuitive interpretation as "and", "or" and "not", and $x \Rightarrow y$ may be read as *if x happens, so does y*. Each AND gate with output node x and input nodes $y_0, \ldots, y_n$ may then be represented, for the completeness and the correctness aspects respectively, as

AND *cmp*

$$x \Rightarrow \bigwedge\{y_0, \ldots, y_n\}$$

AND *crr*

$$\bigwedge\{y_0, \ldots, y_n\} \Rightarrow x$$

and similarly, each OR gate may be represented as

OR *cmp*

$$x \Rightarrow \bigvee\{y_0, \ldots, y_n\}$$

OR *crr*

$$\bigvee\{y_0, \ldots, y_n\} \Rightarrow x$$

where $\bigwedge$ and $\bigvee$ extend $\wedge$ and $\vee$ to sets.

A fault tree may be associated to a set R_{cmp} of *cmp* formulas giving its representation w.r.t. completeness, and to a set R_{crr} of *crr* formulas giving its representation w.r.t. correctness.

It is a relatively straightforward exercise to show that it always holds

Cmp(M)

$$\bigwedge R_{cmp} \wedge f \Rightarrow M$$

Crr(M)

$$\bigwedge R_{crr} \wedge M \Rightarrow f$$

where f is the main failure of the fault tree and M is the Boolean representation of its minimal cut sets; hence

Eq(M)

$$\bigwedge(R_{cmp} \cup R_{crr}) \Rightarrow (M \equiv f)$$

So, from the logical representation of the fault tree and the negation of M, the negation of the main failure follows, i. e. $\neg M$ is a safety condition for the given AND-OR tree.

It is possible to compute M by an algorithm closely associated to property *Cmp(M)* — this consists of applying exhaustively, as rewrite rules, the conditionals in R_{cmp}, starting from f. Since all names in a fault tree are unique, there can be neither cycles nor multiple applications of the same rule. It is routine to prove that the formula thus obtained is equivalent to M.

It is also true in general that, once assumed R_{cmp}, M can be proved to be the weakest formula containing only $wedge$ and vee with the given properties; i. e., for any $wedge$/vee-formula K that satisfies *Cmp(K)* and *Crr(K)*, it always holds

$$R_{cmp} \wedge K \Rightarrow M$$

In fact, K can satisfy *Crr(K)* only if in R_{crr} there are conditionals that allows for the deduction of f from K; but the conditionals in R_{crr} are exactly the converses of those in R_{cmp}; this, together with the fact that M has been obtained by exhaustive application of the latter, ensures the result, allowing us to say that M is the most "liberal" safety condition, i.e. the weakest one.

1.6 Temporal analysis

Basic FTA can be insufficient as failure analysis in presence of temporal constraints. Time may be taken into account in different ways. A possibility [14, 15] is to represent causal consequence, and therefore gates, in a temporal logic, and to define failure analysis in terms of temporal deduction. Another possibility, pursued in [9], is to formalise causal consequence in terms of predicate logic, and to define failure analysis as an algorithm that can handle explicit time information.

1.7 Related work

The semi-formal character of the fault tree method commonly used in industry has plausibly something to do with utilisation in contexts where full system modelling is either infeasible or too costly. Indeed, a common application of fault trees is stochastic — *i.e.* probability assessment of critical events based on probability associated with minimal cut sets. However, formalisation is necessary when we want to derive proof obligations.

This formalisation has taken different paths. There are semantics of fault trees based on system models — [10, 8]. There are formal definitions of fault tree, where the relationship between logic models and system models is shown to be close enough [14, 15, 3]. Some safety analysis tools allow for synthesis of fault trees from system models coupled with fault models, in order to carry out probabilistic failure analysis [2, 12, 11].

The approach presented in [10], particularly focused on dynamic gates and implemented as a tool [6, 5] is based on specification in Z and on translation to Markov models, allowing for an account of the probabilistic aspect. The approach presented in [8, 9], particularly focused on the timing aspect, is based on algebraic modelling of time and events (each associated with a start and an end), on a definition of fault trees in predicate logic, and on a translation to timed Petri nets. A simpler translation of fault trees to generalised stochastic Petri nets can be found in [4].

Logic-based accounts of fault trees are usually based on a temporal logic. This is the case for the formalisation based on Interval Temporal Logic presented in [15] as well as for those based on Mu-calculus proposed in [3]. The verification framework presented in [14] — and implemented [7] — uses fault trees to represent requirements and phase automata to model real-time systems; both formalisms are shown to be representable in a fragment of the Duration Calculus; this makes it possible to model-check the validity of requirements.

1.8 Our approach

The proposed application of failure analysis is extraction of controller specifications expressed in Boolean logic. To this purpose we are going to introduce a fault tree-style representation of system failures that relies, essentially, on decompositional and static gates — in what we call the basic formalism — although capable of capturing some aspect of dynamic behaviour — with what we call the concrete formalism, defined as an extension of the basic one. Time can be important in order to express properties of interlocking systems, but it does not appear convenient to express it explicitly, given the limited expressiveness of the language allowed by the verification tool. For this reason, we are going to keep the temporal aspect into account by replacing the notion of minimal cut set with a stronger one, rather than by using a more expressive logic.

We are also departing from standard fault tree analysis by introducing distinctions between events which are essentially associated with the modelling framework of discrete event systems [13] — notably, the distinction between instantaneous events (hereafter simply *events*) and *conditions* (comparable to what in traditional fault tree terminology are non-instantaneous events); and, further, the distinction between *controllable* and *uncontrollable* events, as well as between *observable* and *unobservable* conditions. These approach is motivated by the prospective goal of making the extraction of fault trees from models inclusive of an environment more efficient.

1.9 Report organisation

In Section 2 we introduce the basic formalism together with its temporal semantics. In Section 3 we introduce our notion of safety analysis. In Section 4 we formalise fault trees and we describe an algorithm for the extraction of safety conditions. In Section 5 we present the concrete formalism that has been defined with end users in mind. In Section 7 we give an example of application.

2 Basic formalism

2.1 Conditions and events

Our approach to high-level failure analysis is based on the distinction between two primitive types of *behaviours* — events and conditions.

$$\begin{array}{lcl} Event & : & type \\ Cond & : & type \end{array}$$

Intuitively speaking, conditions last through time whereas events are sequences of pointwise occurrences. Event occurrences have no duration; moreover, different events may be associated to different timers; therefore, in practice, it may be difficult to say whether occurrences of different events take place at the same time. On the other hand, stating that two conditions hold at the same time does not appear problematic. Therefore, we are not using representation of synchronous events, rather of events taking place while a condition is satisfied — without ruling out the possibility of simultaneity, anyway.

For the nature of the application, we find it convenient to present a formalism that restricts to events having a single occurrence, as well as to conditions that once true never become false. It follows that here events are the same as event occurrences, and that it is possible to define a bijection between events and conditions. We preserve the distinction, however, as it may be useful in practice to tell conditions that define the state of the system apart from events that determine its changes.

Events are classified into either controllable or uncontrollable, depending on whether they are controllable by the controller or controlled by the environment. Similarly, conditions are either observable or unobservable, depending on whether they are observable or not by the controller. An unobservable condition is meant to be totally opaque to the controller, which can never say when it does not hold — even in case this could be inferred theoretically.

Controllable events may be associated, in practice, to system events (often commands) that the controller may allow or not allow to happen. We want to express safety conditions that refer to controllable events, insofar as they say at which time the associated events must not happen in order to avoid a failure, and we want these conditions to be the most liberal compatibly with safety. Time should be expressed qualitatively, in terms of observable conditions. So, in general, a condition may consist of saying that a controllable event must not happen when certain observable conditions hold together.

We are going to introduce behaviours and gates directly from their formal semantics, and to define fault trees, based on this semantics.

2.2 Semantical foundations

In the fault trees that we are considering (SC fault trees, defined further on), parent and child nodes represent behaviours (either events or conditions). Intuitively, behaviours may be thought of as properties of abstract states and gates as transitions between them.

We can essentially distinguish between two sorts of gates. Those of types Cor and $Cand$ are decompositional gates, similar to Boolean operators and can be used in order to build complex conditions. Those of types $C2E$ and $E2C$ are static gates expressing simple temporal relations between input and output.

In the following, we are going to define a semantics for objects and operators, based on their representation in a temporal model. Our notion of time is comparatively unconstrained — we only require that it is an order $<$ on $T' = T \cup \{\infty\}$ where T is a set of finite values with a minimum 0 and ∞ is an infinite value.

$$\begin{aligned}
Cor : &\quad Cond^{+} \rightarrow Cond \\
Cand : &\quad Cond^{*} \rightarrow Cond \\
C2E : &\quad Cond \rightarrow Event \rightarrow T' \rightarrow Boolean \\
E2C : &\quad Event \rightarrow Cond \rightarrow T' \rightarrow Boolean
\end{aligned}$$

Here “$^{+}$” means one or more occurrences, “*” means zero or more occurrences.

Conditions (in our sense) can be represented as non-decreasing Boolean functions on time that become eventually true — *i.e.* as functions that are either always true or have just one step up:

$$StepUp = \{s : T' \rightarrow Bool \mid s\,\infty = True;\ \forall xy : T.\, x \leq y \ \Rightarrow\ (s\,x \Rightarrow s\,y)\}$$

Event occurrences can be represented as time functions giving the occurrence date at current time when the event has already occurred, infinite otherwise.

$$EventOc = \{s : T' \to T' \mid \exists x : T'.\ s\ x = x\ \wedge$$
$$\forall y : T'.\ \mathit{if}\ y < x\ then\ s\ y = \infty\ else\ s\ y = x\}$$

Clearly, for every $e : EventOc$, it always holds

$$\forall t : T'.\ e\ t \neq \infty \Rightarrow e\ t \leq t$$

We can declare $eval$ and $date$ as functions that give a representation of conditions and events in our model, respectively:

$$eval : Cond \to StepUp$$

$$date : Event \to EventOc$$

In the following, we are going to treat $eval$ and $date$ as forms of lifting and hide them, therefore writing $c\ t$ for $eval\ c\ t$ and $e\ t$ for $date\ e\ t$, respectively.

We can define *def* : $Cond$ as a condition that is true at any defined time:

$$\mathit{def}\ t\ =\ True$$

We find it useful to define the following functions, respectively converting conditions to events and vice-versa in obvious ways.

$$cn2ev : Cond \to T' \to T'$$

$$cn2ev\ c\ t\ =\ \mathit{min}\ (\{\infty\} \cup \{x \mid c\ x\ \wedge\ x \leq t\})$$

$$ev2cn : Event \to T' \to Bool$$

$$ev2cn\ e\ t\ =\ (t = \infty)\ \vee\ (e\ t < \infty)$$

2.3 Decompositional gates

$Cand$ and Cor are lifted Boolean operators for conditions, *i.e.* they are constructors for timed Boolean signals.

$$\begin{array}{lcl} Cand(c_1, \ldots, c_n)\ t & = & c_1\ t\ \wedge\ \ldots\ \wedge\ c_2\ t \\ Cor(c_1, \ldots, c_n)\ t & = & c_1\ t\ \vee\ \ldots\ \vee\ c_2\ t \end{array}$$

We can define $Cand()\ =\ \mathit{def}$. Moreover, we will use $\wedge, \vee$ as lifted syntax for $Cand, Cor$, as well.

2.4 Temporal gates and T-constraints

The semantics of temporal gates may be given in terms of abstract relations implying, by axiom, temporal constraints on associated behaviours, here called *T-constrains*.

A gate of type $C2E$ may be thought of as a specification for operators that convert conditions into events. The event may take place at any time when the condition holds. This can be expressed with the following axiom — where the left hand-side is the T-constraint:

$$C2E\ c\ e\ t\ \Rightarrow\ cn2ev\ c\ t\ \leq\ e\ t$$

A gate of type $E2C$ may be thought of as a specification for operators that convert events into conditions. The condition arises when the event occurs. Here the axiom can be:

$$E2C\ e\ c\ t\ \Rightarrow\ cn2ev\ c\ t\ =\ e\ t$$

Note that in both cases the time parameter plays the role of a bound — we are considering the past, not the future.

2.5 SC fault trees

We define an *SC fault tree* to be a fault tree where the only gates are of types $C2E$, $E2C$, $CAnd$ and COr, and where the only controllable events are leaf nodes.

In the following, we are going to discuss the relationship between SC fault trees and the systems they represent, and to introduce a specific form of failure analysis, which, in comparison to standard FTA leads to sets of conditions that are more restrictive and takes implicitly time into account. As a default, by fault trees we will always mean SC fault trees

3 Fault-trees for safe control

3.1 Basic notions

Let H be a fault tree and p a node. We say that the subtree K is *generated* by p whenever it is the greatest subtree that has p as its root.

We say that K is a *direct subtree* of p whenever it is generated by a child node of p.

Given two branches a and b, we say that h is an *initial sub-branch* of k whenever, assuming branches are represented as lists starting from the origin, a is a non-empty prefix of b.

We say that K is an *initial subtree* of H whenever, for every branch a of H, K contains an initial branch of a.

3.2 Deterministic subtrees

We say that a fault tree H is a *deterministic tree* whenever for each OR node, it includes one and only direct subtree. Clearly, determinism is inherited by subtrees.

Let H be a fault tree. We say that K is a *main deterministic subtree* of H whenever K is a maximal subtree such that it is deterministic.

A deterministic tree is equivalent to an AND tree, since all its OR nodes have a single child. So, main deterministic subtrees can represent maximal AND subtrees.

Every fault tree can be represented by the set of its main deterministic subtrees. Intuitively speaking, these are the processes leading to the main failure, and may be regarded as possible explanations.

3.3 Causal sets

Given a deterministic tree, we will call the set of its leaves the *causal set* associated with it. Given the fact that a deterministic tree is essentially and AND tree, causal sets may be considered as special cases of minimal cut sets.

We say that a causal set is *controllable* whenever all of its elements are either conditions or controllable events, and at least one is a controllable event.

Intuitively, controllable causal sets may be associated with safety conditions (with their negation, to be more precise).

3.4 Dead ends

Let H be a fault tree, J a main deterministic subtree of H, and K a subtree of J generated by a node $p \in J$. We say that K is a *dead end* of H whenever either (A) p is an unobservable condition, (B) K does not contain any controllable event.

The notion of dead end is needed in order to capture the idea that there are parts of the system for which either a controller can do nothing (case B) or there is ineliminable uncertainty about whether relevant conditions may hold (case A). In case a dead end is also a main deterministic subtree, it is clearly impossible to guarantee safety of the system, since there is a causal set that cannot be affected by any controller.

3.5 Control tree

We will call *control tree* of a fault tree H the greatest initial subtree K such that it has the same number of main deterministic subtrees as H and has no dead ends, if this exists; otherwise, we will say that the control tree does not exist, or is empty.

In order to make this notion more accessible, we may consider the following definition, extending that of dead end.

Let H be a tree and K a subtree generated by a node $p \in H$. We say that K is a *dead subtree* if either (A) p is an unobservable condition, (B) K does not contain any controllable event, (C) p is an OR node and at least one of its direct subtrees is dead, or (D) all of its direct subtrees are dead.

Now, it is not hard to see that a fault tree has no dead ends if and only if it has no dead subtrees. Therefore, the control subtree is the same as the greatest initial subtree which does not include any dead subtree.

Intuitively speaking, the control tree represents the same processes as the fault tree from which it is extracted, only up to the point these processes are controllable and therefore relevant from the point of view of controller specification. In case the fault tree allows for processes that are not controllable, there is no such thing and the system is inherently unsafe.

3.6 Pruning

Let H be a fault tree. We say that K is a *pruning* of H whenever, for a main deterministic subtree J of H, K is an initial subtree of J, and all of its leaves are behaviour nodes. We will say that a pruning is controllable whenever its causal set is.

Given the fact that all leaves in a fault tree are behavioural, prunings always include main deterministic subtrees.

Intuitively speaking, given a main deterministic subtree, each pruning represents a different temporal cut of the process leading to the main failure. Causal sets of the prunings can represent abstract states of the system — where a state can be interpreted as a set of behaviours holding at the same time.

As to the relationship between prunings of a fault tree H and those of its control subtree K, it is not hard to see that

(A) for each pruning I of H, the intersection of I with K, $J = I \cap K$, is a pruning of K; and

(B) every pruning of K can be obtained in that way.

3.7 Safety Control Analysis

Given an SC fault tree, we will call *control sets* (*CS*) the causal sets of the controllable prunings of the control subtree. We may refer to the extraction of the CS set from a tree as *safety control analysis* (*SCA*).

We are interested in the control subtree as the part of the system that can be actually controlled, and in the controllable prunings as those associated with controllable causal sets, and therefore with safety conditions. As to the relationship between FTA and SCA, one may note that, given a control tree, the causal sets that contain only controllable events can represent the minimal cut sets.

SCA gives a result that tends to be stronger than standard FTA, insofar as it takes qualitative time into consideration. Unfortunately, if the number of minimal cut sets can grow exponentially in the number of the OR nodes, the number of control sets can grow exponentially already in the number of the AND ones.

4 Formalisation

4.1 Inference system

Our formal representation of fault trees is based on a language L that includes a set of names for behaviours (events and conditions), operators $E2C, C2A, Cand, Cor$ for gate relations, and the defined operator $ev2cn$.

We will call *temporal formula* in L any expression of type $T' \to Bool$. We will call *transition* any instantiation of gate relation with behaviour names. We use k_i, with $i : Nat$, as a metavariable for either a condition name c or for a condition expression $ev2cn\ e$, where e is an event name. We will call *state formula* any expression of form $Cand[k_1, \ldots k_n]$.

We can define an inference system to derive safety conditions in terms of provably correct rewrite rules associated to $E2C, C2E, Cor$ transitions and applicable to sets of state formulas. Each rule can be derived from a transition, by instantiation of one of the following rule schemas. Let S be a set of state formulas. We will write $S + \alpha$ for $S \cup \{\alpha\}$.

Event Elimination: given a transition $C2E\ c\ e$, derive

$$S\ +\ Cand(\overline{k_1},\ ev2cn\, e,\ \overline{k_2})\ \Longrightarrow\ S\ +\ Cand(\overline{k_1},\ c,\ \overline{k_2})$$

Condition Elimination: given a transition $E2C\ e\ c$, derive

$$S\ +\ Cand(\overline{k_1},\ c,\ \overline{k_n})\ \Longrightarrow\ S\ +\ Cand(\overline{k_1},\ ev2cn\, e,\ \overline{k_n})$$

Or Elimination: given a transition $c = Cor\ [c_1, \ldots, c_n]$, derive

$$S\ +\ Cand(\overline{k_1},\ c,\ \overline{k_2})\ \Longrightarrow\ S\ +\ Cand(\overline{k_1},\ c_1,\ \overline{k_2})\ +\ \ldots\ +\ Cand(\overline{k_1},\ c_n,\ \overline{k_2})$$

4.2 Formal states

State formulas may be used to represent causal sets, which then, according to our semantical interpretation of $Cand$, are sets of conditions holding together at a certain time. More precisely, we may say that a state formula $\alpha = Cand(k_1, \ldots k_n)$ (does not hold) holds at a time t when $k_1\ t\ \wedge\ \ldots\ \wedge\ k_n\ t$ (does not hold) holds. If α (does not hold) holds at any finite time, we can simply say that α (does not hold) holds, and replace $k_1, \ldots, k_n$ with Boolean variables to obtain the *Boolean translation* of α.

Each component of a fault tree can be represented by a transition. Note that we have assumed behaviour names to be individual constants rather than variables, and therefore we do not need existential quantification. We will call the set of transitions associated with a fault tree H, together with a partition of the event names between controllable and uncontrollable, and a partition of the condition names between observable and unobservable, the *logical representation* of H.

We are going to treat sets of state formulas like disjunctions, saying that a set of state formulas (does not hold) holds (at t) whenever one of its elements does. Each rewrite rule follows logically from the associated transition, in the sense that if the left-hand side set of the rule holds at a certain time, then the right-hand side set does as well. The proof follows from the T-constrains associated to the transitions.

Indeed, it follows for each transition, by the T-constraint associated with it, that the logical relationship between the left hand-side α and the right hand-side β of the associated rewrite rule is actually a weak form of equivalence:

RA

$$\exists t : T.\, \alpha\, t \;\equiv\; \exists t : T.\, \beta\, t$$

Rewrite rules may be used to trace backwards the possible causes leading to the main failure, corresponding to the completeness aspect of FTA; on the other hand, converse rewrite rules can be used to derive possible effects from causes, corresponding to the correctness aspect of FTA. We take the notion of possibility in the correctness aspect to be purely implicit. Time intervals implicitly associated to $C2E$ relations are not kept into account quantitatively, but rather qualitatively, using the SCA approach.

In order to represent formally the causal sets of the prunings, we can now introduce a notion of *formal state* inductively, as follows.

The *final state* of a fault tree H with main failure f can defined as $ev2cn\ f$ (in fact equivalent to $Cand(ev2cn\ f)$).

The *set of all formal states* of H is the smallest set of state formulas which includes the final state and is closed with respect to the application of the rules derived from the logical representation of H.

It is a matter of routine to show that for each pruning, its causal set is represented by a formal state. A *controllable formal state* is a formal state that, according to the logical representation, represents a controllable causal set.

4.3 Declarative SCA

It is now possible to specify SCA using the notion of formal state. The task consists of finding the controllable formal states of the control tree, as follows.

Given a fault tree H,
(A) extract from H the control subtree K;
(B) define the logical representation K' of K;
(C) compute the set S of all formal states fro K';
(C) compute the set T of all controllable formal states from S and K'.

It is a matter of routine to show, using the RA weak equivalence for each rewrite rule, that the result of formal SCA holds at a finite time, whenever the main failure may take place at some finite time, as well.

Let T be the result of SCA and $T' = \{s_1, \ldots, s_n\}$ be the set of Boolean translation of the elements of T. Assuming at the semantical level that the fault tree represents all the possible ways in which the main failure f may take place in the system, the safety condition sc relative to f can be defined, equivalently, either as

$$\neg(s_1\ \vee\ \ldots\ \vee\ s_n)$$

or as

$$\neg s_1 \;\wedge\; \ldots \;\wedge\; \neg s_n$$

The proof that sc is actually a safety condition relative to f, i.e. that

$$sc \;\Rightarrow\; \forall t : T.\; \neg\; (ev2cn\; f)$$

is quite straightforward, given the above. The proof that sc is the weakest safety condition, may be given along lines similar to those used for the analogous requirement in FTA.

4.4 Procedural SCA

In the previous section we have used a declarative approach, defining logical representation and formal states, and then defining the result of SCA as controllable formal states.

In contrast, here we will give a procedure such that it can be used, starting from the final state, to build a set of state formulas that in fact converges to the set of the controllable formal states.

Moreover, rather than pre-processing the fault tree in order to get the control subtree, we add a semantical rule. Dead ends can be neutralised by making the worst possible assumption, in terms of safety, about their top node. This can be obtained by applying the following rule.

DE Elimination: if the behaviour name in k_i is member of a dead end, derive

$$S \;+\; Cand(\overline{k_1},\; k_i,\; \overline{k_2}) \;\Longrightarrow\; S \;+\; Cand(\overline{k_1},\; \overline{k_2})$$

Then the algorithm goes as follows.

1. Compute the logical representation of the fault tree.
2. Initialise the state set S to the singleton of the final state.
3. Repeat until saturation: for each state formula α in S

- compute the applicable rules (selected rules);
- apply each of the selected rules, adding the result to the state set
- delete α from S, if all the selected rules have been applied, unless α is controllable (i.e. represents a controllable causal set, according to the logical representation), and all the selected rules of α are instances of Condition Elimination.

Showing that this algorithm satisfies the specification, i.e. converges to the controllable formal states, is a matter of routine. The fact that the order of execution does not matter, follows from the fact that there can be just an applicable rule for each behaviour name — apart from DE Elimination, which may be delayed at wish, since its semantical precondition holds only when it extends to child nodes.

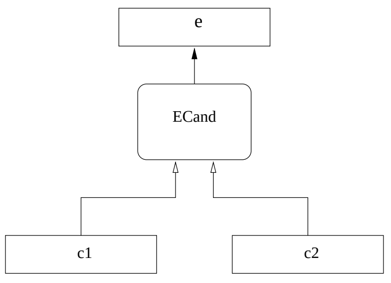
5 Concrete formalism

In the concrete formalism, events, conditions and gate types are graphically distinguished. Extra-logical information about events and conditions is conveyed by means of colours — events are either controllable or non controllable, conditions are either observable or non observable.

Noticeably, in the concrete formalism we tend to avoid the use of $Cand$; rather, we introduce the following gate types which combine $Cand$ with other ones.

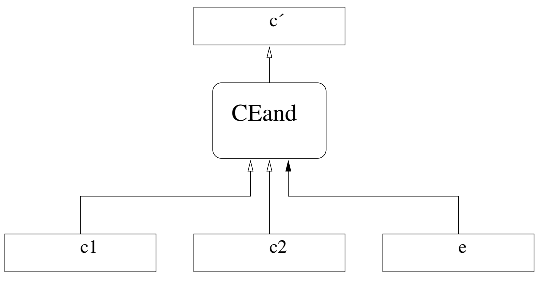
$$\begin{array}{rr} ECand: & COND^{+} \rightarrow EVENT \rightarrow T' \rightarrow Bool \\ CEand: & COND^{*} \times EVENT \rightarrow COND \rightarrow T' \rightarrow Bool \end{array}$$

$ECand$**:** The output event occurs when some input condition becomes true while the other ones are true — this essentially expresses causal dependency of the output event on input conditions regardless of the order in which the latter become true.



$$ECand\,(c_1,\ldots,c_n)\;e \;=\; C2E\;(Cand\,(c_1,\ldots,c_n))\;e$$

$CEand$**:** The output condition becomes true as soon as the input conditions are true and the input event occurs — this essentially expresses causal dependency of the output on an event that takes place while a condition holds. Note that this relation involves the temporal order of inputs, and therefore has dynamic character; however, e cannot be further developed when there are side conditions. The use of this operator is recommended for situations in which the relation between the input conditions and the input event is not a cause-effect one.



$CEand\,(e,\bar{c})\;c' \;=$
$if\; \bar{c} \neq \emptyset$
$then\; ECand\,(\bar{c})\; e\; and\; E2C\; e\; c'$
$else\; E2C\; e\; c'$

6 Conclusion and further work

We have presented a formal method that associates top-down analysis of system failures in the style of fault trees, allowing for extraction of safety conditions, together with system modelling based on a temporal semantics in the style of discrete event systems [13].

The application VeriFT, built as part of the project, includes an implementation of SCA in Haskell and a graphic interface implemented using Java and uDrawGraph. Further work should involve addressing open issues, particularly with respect to the expressive power of the formalism and possible extensions.

There are still some aspects related to time that need to be considered. As to the representation of temporal constraints in the system model, we expect to handle them explicitly by introducing events for timer start and timeout.

We would like to be able to represent the fact that, in order to preserve safety, some actions *have* to be taken — whereas at present the only safety conditions that we can use are action-disabling ones.

Finally, although our analysis does not cover stochastic aspects, we think it may be practically useful to introduce a weaker form of safety — associated to conditions that make the system safe in "most cases", or under ordinary circumstances. To this purpose, we have been considering the introduction of "rare event" as a special type of event.

7 An example

7.1 The problem

Figure 1 is a fault-tree representing an accident — originally a train derailment within an interlocking system, disguised as asked by our industrial partner *Thales*.

Here the accident consists of a cat eating a bird. The informal reasoning goes that if there is a bird free in the room and there is a cat as well, the bird might end up eaten. The act of opening the cage sets the bird free. The act of opening the room door while the cat is behind it, allows the cat to enter the room. The most liberal discipline for safety, consists of avoiding to perform one of the two acts under any course of events started by the other one. We will now illustrate how safety conditions can be extracted procedurally in this simple case.

7.2 Deriving safety conditions

The logical representation of the fault-tree contains the following transitions. Conditions and event names are abbreviated to their acronyms with respect to the labels used in Fig. 1. We will write e' for $ev2cn\ e$.

$$ECand\ (cir, \mathit{bf})\ ceb \tag{1}$$
$$E2C\ cer\ cir \tag{2}$$
$$E2C\ oc\ \mathit{bf} \tag{3}$$
$$ECand\ (do, \mathit{cbd})\ cer \tag{4}$$
$$E2C\ od\ do \tag{5}$$

The initial set contains only the final state:

add s0:

$$Cand(ceb')$$

We can apply to *s0* an instance of Event Elimination derived from relation 1 (after expansion with the definition of $ECand$). Moreover, *s0* can be deleted from the set, since it is not controllable and all the selected rules have been applied.

delete s0 (all rules applied, uncontrollable).

add s1:

$$Cand(cir,\ \mathit{bf}\text{'})$$

Now it is possible to apply Condition Elimination derived from relation 3 to *bf'*.

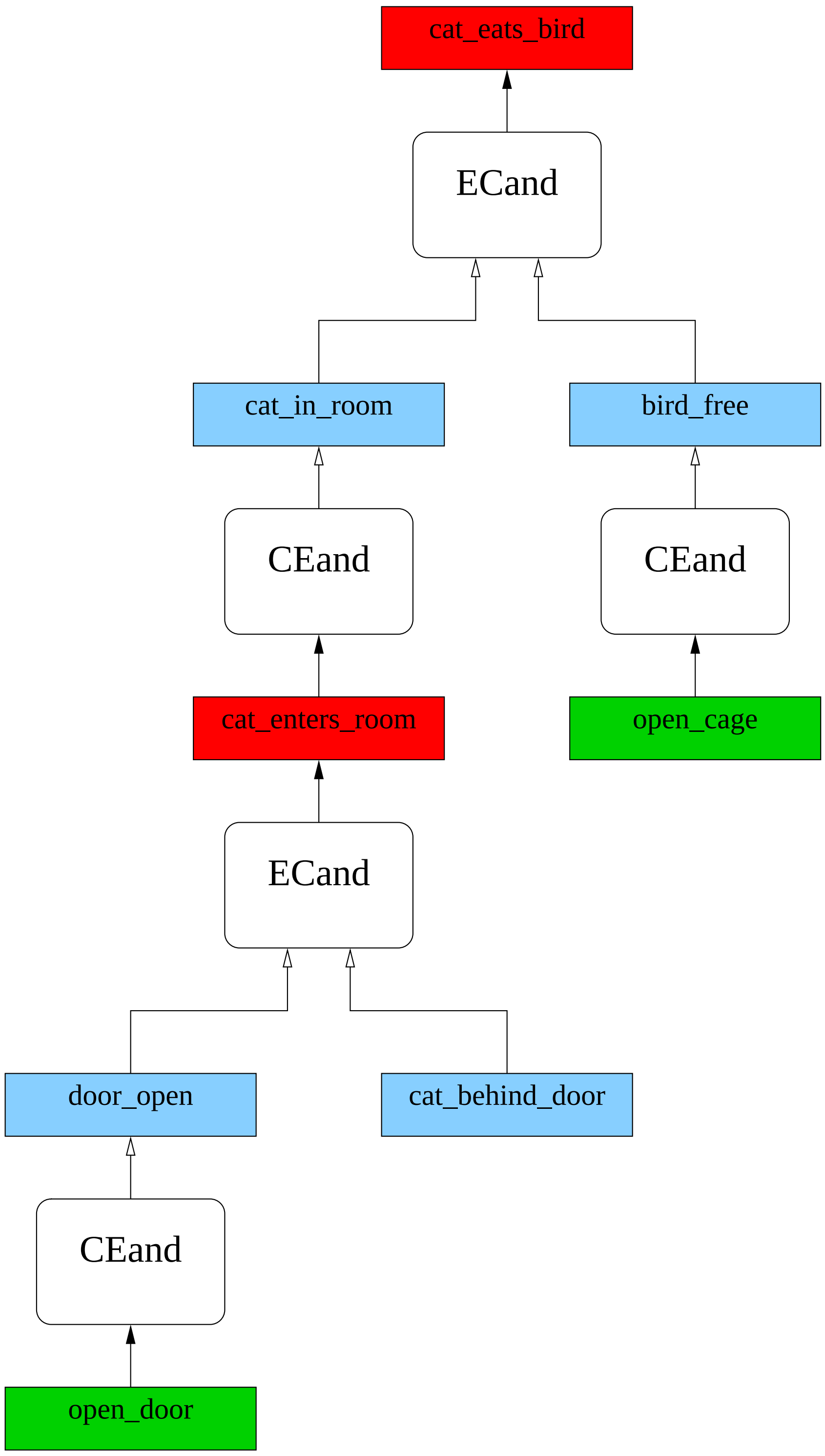


Figure 1: A derailment example

add s2:

$$Cand(cir,\ oc')$$

This is a controllable state. State *s1* can also be transformed by Condition Elimination derived from relation 2.

delete s1 (all rules applied, uncontrollable).

add s3

$$Cand(bf,\ cer')$$

This is not a controllable state, as cer is not controllable, and indeed it can be further transformed — by Event Elimination (from 4), using also DE Elimination for cbd and simplifying.

add s4:

$$Cand(bf,\ do')$$

We can apply Condition elimination (from 5).

add s5:

$$Cand(bf,\ od')$$

This is also a controllable state, and this pair cannot be further refined. On the other hand, *s2* as well as *s3* can be refined in another way.

delete s3 (all rules applied, uncontrollable).

add s6:

$$Cand(cer',\ oc')$$

This can be transformed using Event Elimination. Alternatively, from s4 by Condition Elimination.

delete s6 (all rules applied, Event elimination applied);
delete s4 (all rules applied, uncontrollable).

add s7:

$$Cand(do,\ oc')$$

From this we can also get the last one.

add s8:

$$Cand(od,\ oc')$$

States *s7* and *s8* are controllable as well. Therefore the final set is $\{s2, s5, s7, s8\}$. Negating the Boolean translation of each element and taking the conjunction amounts to saying

Don't open the bird cage either when the cat is in the room or when the room door is open, don't open the room door when the bird is free, and don't open the two doors at the same time.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1995. 1.2

[2] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In *Proc. 4th European Dependable Computing Conference*, volume 2485 of *LNCS*, pages 19–31. Springer, 2002. 1.7

[3] G. Bruns and S. Anderson. Validating safety models with fault trees. In J. Gorski, editor, *Computer Safety, Reliability and Security '93*, pages 21–30. Springer, 1993. 1.5, 1.7

[4] D. Codetta-Raiteri. The conversion of dynamic fault-tree to stochastic Petri nets, as a case of graph transformation. *Electronic Notes in Theoretical Computer Science*, 127:45–60, 2005. 1.7

[5] D. W. Coppit and K. J. Sullivan. Formal specification in collaborative design of critical software tools. In *Hight Assurance System Engineering '99*, 1999. 1.7

[6] J. B. Dugan, K. J. Sullivan, and D. W. Coppit. Developing a high-quality software tool for fault tree analysis. In *Software Reliability Engineering '99*, 1999. 1.7

[7] J. Faber. Fault tree analysis with Moby-FT. Technical report, Universitaet Oldenburg, 2005. 1.7

[8] J. Gorski and A. Wardzinski. Formalising fault trees. In *Safety Critical Systems Symposium '95*, 1995. 1.3, 1.7

[9] J. Gorski and A. Wardzinski. Timing aspects of fault tree analysis. In T. Anderson F. Redmill, editor, *Safer Systems*, pages 231–244. Springer, 1997. 1.6, 1.7

[10] R. Manian, D. W. Coppit, K. J. Sullivan, and J. B. Dugan. Bridging the gap between systems and dynamic fault tree models. In *Reliability and Maintainability Symposium '99*, pages 105–111, 1999. 1.7

[11] M. McKelvin, G. Eirea, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In *5ft ACM International Conference on Embedded Software*, pages 237–246, 2005. 1.7

[12] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner. Analysis and sysnthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71:229–247, 2001. 1.7

[13] P.J.G. Ramadge and W. M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77(1), pages 81–98, 1989. 1.2, 1.8, 6

[14] A. Schaefer. Combining real-time model-checking and fault tree analysis. In *Formal Methods '03*. Springer, 2003. 1.2, 1.6, 1.7

[15] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Integrated Design and Process Tehcnology '02*, 2002. 1.6, 1.7

[16] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000. 1.2

[17] W. E. Vesely and et al. *Fault Tree Handbook*. Nureg 0492, US Nuclear Regulatory Commission, 1981. 1.3