Retrenchment, and the Generation of Fault Trees for Static, Dynamic and Cyclic Systems ⋆

Richard Banach[1], Marco Bozzano[2]

[1] School of Computer Science, University of Manchester, Manchester M13 9PL, UK
banach@cs.man.ac.uk
[2] ITC-IRST, Via Sommarive 18, Povo, 38050 Trento, Italy
bozzano@irst.itc.it

Abstract. For large systems, the manual construction of fault trees is error-prone, encouraging automated techniques. In this paper we show how the retrenchment approach to formal system model evolution can be developed into a versatile structured approach for the mechanical construction of fault trees. The system structure and the structure of retrenchment concessions interact to generate fault trees with appropriately deep nesting. The same interactions fuel a structural approach to hierarchical fault trees, allowing a system and its faults to be viewed at multiple levels of abstraction. We show how this approach can be extended to deal with minimisation, thereby diminishing the post-hoc subsumption workload and potentially rendering some infeasible cases feasible. The techniques we describe readily generalise to encompass timing, allowing glitches and other transient errors to be properly described. Lastly, a mild generalisation to cope with cyclic system descriptions allows the timed theory to encompass systems with feedback.

1 Introduction

Reliability analysis of complex systems traditionally involves a set of activities which help engineers understand the system behaviour in degraded conditions, that is, when some parts of the system are not working properly. These activities have the goal of identifying all possible hazards of the system, together with their respective causes. The identification of hazards is a necessary step for safety-critical applications, to ensure that the system meets the safety requirements that are required for its deployment and use.

Among the safety analysis activities, a very popular one is Fault Tree Analysis (FTA) [30]. It is an example of deductive analysis, which, given the specification of an undesired state –usually a failure state– 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*, that is, a graphical representation of the logical interrelationships of the basic events that lead to the undesired state.

The manual construction of fault trees relies on the ability of the safety engineer to understand and to foresee the system behaviour. As a consequence, it is a time-consuming and error-prone activity, and may rapidly become impractical in case of

⋆ Work partly supported by the E.U. ISAAC project, contract no. AST3-CT-2003-501848.

large system models. Therefore, in recent years there has been a growing interest in formally based techniques to automate the production of fault trees [15, 13].

The starting point is our previous work relating retrenchment [5, 6, 7, 8, 9, 10, 3] and formal system model evolution [4]. Namely, in [4] we showed how retrenchment, as opposed to conventional refinement, can provide a formal account of the relationship between the abstract system model, that is the model of the system in nominal conditions, and the concrete system model, that is the model enriched with a description of the envisaged faults the system is designed to be robust against.

In this paper we show how retrenchment can be developed into a versatile structured approach for the mechanical construction of fault trees. Building on the ideas sketched in [4], where we exemplified the generation of a fault tree on a two-bit adder example, in this paper we show how the simulation relation of retrenchment can be used to systematically derive fault trees built upon the system structure. This is achieved by exploiting the structure of retrenchment concessions, using suitable notions of composition to gather the degraded cases into the concession of a composed retrenchment. We show how these techniques can be readily generalised in order to deal with issues like timing and cycles, thus paving the way for the analysis of dynamic systems and systems with feedback. Finally, we show how the interactions between the system structure and the structure of concessions yield a structural approach to hierarchical fault trees, allowing a system and its faults to be viewed at multiple levels of abstraction.

The techniques we present in this paper improve over the ones discussed in [13], in that they allow the mechanical generation of fault trees built upon the system structure, which are more informative than the flat (two-level) fault trees of [13]. Furthermore, we demonstrate the potential of our approach by exemplifying how these techniques can be fruitfully adapted to address the problem of generating the minimal cut sets of a fault tree. We show that, by annotating the generated subtrees with suitable minimisation directives, it is possible to perform some minimisations locally, thereby diminishing the post-hoc, brute-force subsumption workload of traditional minimisation algorithms.

The rest of the paper is structured as follows. In Section 2 we review retrenchment and relevant notions of composition of retrenchments. In Section 3 we present our retrenchment directed approach to the generation of hierarchical and structured fault trees on a running example. In Section 4 we show how the structured analysis can be modified to reduce the work of finding the minimal cut sets of some fault condition. In Section 5 we extend the method to deal with internal state in the subsystem being treated, which is relatively straightforward as long as the subsystem remains acyclic, and in Section 6 we discuss the issues raised by cyclicity and feedback. Finally, in Section 7 we discuss some related work and we outline some conclusions.

2 Systems, Retrenchments and Compositions

In this paper we describe systems using input/output transformers. So in general, a (sub)system will consist of a collection of I/O relations, each describing the behaviour of a component, and with (sub)system structure expressed by the identification of predecessor component outputs with successor component inputs; obviously some inputs and outputs remain free to allow communication with the environment. We can write

such components using a relational notation $Thing(i, o)$, where i and o can be tuples, eg. $i = \langle x, y, z \rangle$ in the case of multi-input/output components.

Retrenchment [5, 6, 7, 8, 9, 10, 3], was introduced to provide a formal vehicle for describing more flexible model evolution steps than the usual technique for formal system development, refinement, conventionally allows. Since refinement was conceived with the desire to ensure that the next model conformed to properties of its predecessor, while moving towards greater implementability, it is no surprise that not all model evolutions that one might conceivably find useful fall under its scope.

In this paper, it is the simulation relation of retrenchment which does the work. This can be expressed as follows. Suppose we have two systems *Abs* and *Conc*, and suppose $Op_A(i, o)$ and $Op_C(j, p)$ are two corresponding operations, aka component behaviours, in *Abs* and *Conc* respectively.[3] A retrenchment simulation between them is given by:

$$W_{Op}(i,j) \wedge Op_C(j,p) \wedge Op_A(i,o) \wedge (O_{Op}(o,p,i,j) \vee C_{Op}(o,p,i,j))$$

Here W_{Op}, O_{Op}, C_{Op} are the *within*, *output*, *concedes* relations for the pair of operations Op. The within relation W_{Op} defines the remit of the retrenchment; while the output and concedes relations describe what are to be considered 'normal' and 'deviant' aspects of the relationship between Op_A and Op_C. The aggregate of all the relevant relations for all corresponding operation pairs is collectively called the *retrenchment data* for the particular retrenchment between *Abs* and *Conc* that we have in mind.

To consider large systems, we need mechanisms to express hierarchy and composition. Fortunately these are straightforward. To express hierarchy, it will be sufficient to decompose the concession into a number of cases covering distinct fault possibilities: $C_{Op} \equiv C_{Op,1} \vee C_{Op,2} \vee \ldots \vee C_{Op,n}$. So C_{Op} expresses the high level view while the $C_{Op,k}$ give a more detailed lower level perspective.

For composition, we need sequential and parallel composition mechanisms. Fortunately these are both straightforward, and similar to each other. Given $Op1$ and $Op2$, assuming the outputs of $Op1$ can be identified with the inputs of $Op2$, their sequential composition $Op1;2$ is the relational composition $Op1 \fatsemi Op2$. If now both $Op1$ and $Op2$ come in abstract and concrete versions, related by retrenchment data $W_{Op1}, O_{Op1}, C_{Op1}$ and $W_{Op2}, O_{Op2}, C_{Op2}$ respectively, then $Op1;2_A$ and $Op1;2_C$ will be related by retrenchment data:

$$W_{Op1;2} = W_{Op1} \qquad (\textit{provided} \quad (O_{Op1} \vee C_{Op1}) \Rightarrow W_{Op2})$$
$$O_{Op1;2} = O_{Op1} \fatsemi O_{Op2} \qquad C_{Op1;2} = O_{Op1} \fatsemi C_{Op2} \vee C_{Op1} \fatsemi O_{Op2} \vee C_{Op1} \fatsemi C_{Op2}$$

where $\vee$ is relational union. Parallel composition is even easier. Assuming this time that $Op1$ and $Op2$ act on independent sets of variables, and using $|$ to denote parallel composition (which, in terms of logic, is just conjunction), the rules are:

$$W_{Op1|2} = W_{Op1} | W_{Op2}$$
$$O_{Op1|2} = O_{Op1} | O_{Op2} \qquad C_{Op1|2} = O_{Op1} | C_{Op2} \vee C_{Op1} | O_{Op2} \vee C_{Op1} | C_{Op2}$$

We see the strong analogy between the two. Moreover, these interact cleanly both with each other, and with the hierarchy mechanism. Thus if C_{Op1} is a disjunction of n terms,

[3] Correspondence of operations in *Abs* and *Conc* is a meta level concept, which we indicate by using the same name for the operation in the two systems, or by other convenient means.

and C_{Op2} of m terms, the lower level versions of $C_{Op1;2}$ and $C_{Op1|2}$ have $mn + m + n$ terms, corresponding to the substitution of the low level forms into $C_{Op1;2}$ or $C_{Op1|2}$ respectively.

3 Hierarchy and Fault Tree Structure in a Running Example

In Fig. 1 we see a small circuit which will serve as a running example. At a high level it is a black box called *Fred* with two inputs $I1, I2$ and two outputs $O1, O2$. At a low level, it is a circuit in which signals flow from left to right, elements $A1, A2, A3$ are adders, and $F1, F2, F3$ are two-output fanout nodes. We assume that all signals are of a fixed finite number of bits, and that the adders do cutoff addition (which is to say that any value greater than or equal to the maximum representable one is output as the maximum, and there is no overflow). The number of bits is assumed sufficiently large that the cutoff effects do not occur in the examples we treat. The two diagrams in Fig. 1 represent a descent of one level in a hierarchical description of (part of) a large system.

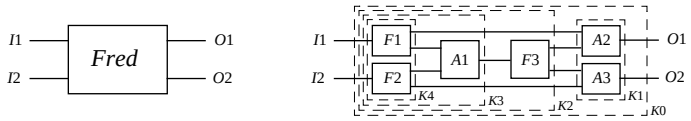

Fig. 1. A subsystem *Fred* and its internal structure.

We turn to the internal structure of *Fred*. For the time being, all elements are stateless, and all circuits are acyclic. Such circuits possess a parsing which builds them up via sequential and parallel composition. In general there will be several such parsings. We choose the one in which the elements closest to the inputs are the most deeply nested: it can be derived mechanically from a definition of the circuit in terms of elements and connections, or supplied manually. Such a structure is in sympathy with a top-down fault analysis starting at the outputs. For *Fred* the structuring is illustrated in $K0$-$K4$.

Introducing names for the internal variables implicitly, the ideal $Fred_A$ model is given by fanout component relations: $F1_A(I1, \langle a1, a2\rangle) \equiv a1 = a2 = I1$ (similarly for $F2_A(I2, \langle a3, a4\rangle)$ and $F3_A(a5, \langle a6, a7\rangle)$); and adder component relations, given by: $A1_A(\langle a2, a3\rangle, a5) \equiv a5 = a2 + a3$ (similarly for $A2_A(\langle a1, a6\rangle, O1), A3_A(\langle a7, a4\rangle, O2)$).

Fred's potentially faulty behaviour, model $Fred_C$, is given using renamed variables for clarity. Thus the external inputs/outputs are $J1, J2$ and $P1, P2$ respectively, and the internal variables $a1$-$a7$ become $c1$-$c7$. We assume that only the fanouts can have faults, and that these are simply 'stuck_at_0' faults on one or other output, signalled by the truth of additional free boolean variables $F1.c1$ ($F1$ output $c1$ 'stuck_at_0') etc. *We assume (purely for simplicity) that only one fault can be active in any component (at any time).* Thus while the adders $A1_C, A2_C, A3_C$ in $Fred_C$ are given by mere transliterations of the $A1_A, A2_A, A3_A$ relations above to J, P, c variables, the fanouts need full redefinition, eg.:

$$F1_C(J1, \langle c1, c2\rangle) \equiv (F1.c1 \Rightarrow c1 = 0) \wedge (F1.c2 \Rightarrow c2 = 0) \wedge \mathit{ONE} \quad \mathit{ELSE\_IDEAL}$$

In this, $\mathit{ONE} = \neg(F1.c1 \wedge F1.c2)$ and $\mathit{ELSE\_IDEAL}$ represents the transliteration of $F1_A$ to J, P, c variables, when not overridden by the faulty behaviour of the preceding terms.

The ideal and faulty *Fred* models are related by a retrenchment. It will be sufficient to write down the retrenchment data for just the components, since the data for the overall system will emerge as needed from the fault tree analysis below. For the adders, assumed fault-free, we have for $A1$:

$$W_{A1}(\langle a2, a3\rangle, \langle c2, c3\rangle) \equiv \mathsf{true}$$
$$O_{A1}(a5, c5, \langle a2, a3\rangle, \langle c2, c3\rangle) \equiv (c5 = c2 + c3)$$
$$C_{A1}(a5, c5, \langle a2, a3\rangle, \langle c2, c3\rangle) \equiv \mathsf{false}$$

with similar things for $A2, A3$; a particular consequence of this is that occurrences of C_A terms can be dropped below. For the fanouts, we need the more complicated:

$$W_{F1}(I1, J1) \equiv \mathsf{true}$$
$$O_{F1}(\langle a1, a2\rangle, \langle c1, c2\rangle, I1, J1) \equiv (c1 = c2 = J1)$$
$$C_{F1}(\langle a1, a2\rangle, \langle c1, c2\rangle, I1, J1) \equiv (F1.c1.0 \wedge c1 = 0 \wedge c2 = J1) \vee (F1.c2.0 \wedge c1 = J1 \wedge c2 = 0)$$

In C_{F1}, following Section 2, we call the two disjuncts $C_{F1,c1}$ and $C_{F1,c2}$ respectively, i.e. $C_{F1} = C_{F1,c1} \vee C_{F1,c2}$. Similar things hold for $F2, F3$. Note that the abstract system is not mentioned in the body of the retrenchment data; it is not needed in this application.

With these ingredients, and a given top level event (TLE), we show how the retrenchment data drive a structured fault analysis. First, if it is of interest to check whether the TLE can arise via fault-free behaviour, it is sufficient to check whether the TLE will unify with O_{Fred}. This is easy to calculate from the assumed parse $K0$-$K4$ and the rules of Section 2, since we will assume that for all components, correct working is given by a total function, and even incorrect working is a total relation.[4] Second, we proceed downward through C_{Fred}, decomposing step by step, eliciting the consequences of composition and of local structure, and deriving a *resolution tree* for all possible ways of satisfying the TLE within the constraints. Values of variables once assigned, remain in force as we descend unless we backtrack past the point of assignment, and once the input values have been reached, any remaining uninstantiated variables can be instantiated within the constraints that hold, case by case, to confirm overall consistency. Now we consider the specific TLE: $J1 = J2 = P1 = 1$ (with $P2$ regarded as irrelevant). It is easy to check that this does not satisfy O_{Fred}. The analysis then proceeds as follows.

TLE: $K0 = K2 \fatsemi K1$, so $C_{K0} = O_{K2} \fatsemi C_{K1} \vee C_{K2} \fatsemi O_{K1} \vee C_{K2} \fatsemi C_{K1}$. Since $K1$ is nearest the outputs, and we are working backwards through *Fred*, we decompose $K1$ first, i.e. we decompose O_{K1} and C_{K1}. Since $K1 = A2|A3$ and adders don't fail, C_{K1} is false, reducing C_{K0} to $C_{K2} \fatsemi O_{K1}$, while $O_{K1} = O_{A2}|O_{A3}$. Now O_{A3} merely imposes existential constraints on $P2, c7, c4$ such that $A3(\langle c7, c4\rangle, O2)$ holds; we put these to one side since the TLE does not constrain them further. O_{A2} demands that $c1 + c6 = 1$ (among other things). There are two ways to satisfy this, namely $c1 = 0 \wedge c6 = 1$ or $c1 = 1 \wedge c6 = 0$, giving a top level disjunction into **TLE.L** or **TLE.R** for $C_{K2} \fatsemi O_{K1}$.

TLE.L: Since $c1$ and $c6$ are outputs of $K2$, we next decompose $C_{K2} = C_{K3;F3} = O_{K3} \fatsemi C_{F3} \vee C_{K3} \fatsemi O_{F3} \vee C_{K3} \fatsemi C_{F3}$. Now $C_{F3} = C_{F3,c6} \vee C_{F3,c7}$, and $C_{F3,c6}$ is inconsistent with $c6 = 1$. Also O_{K3} forces $c5 = 2$, inconsistent with $c6 = 1$ too, so the terms

[4] Similarly, we assume that O_{K4}-O_{K2} can be evaluated immediately from $J1, J2$ when needed. In more general cases, a backwards derivation might be required for some O terms.

containing these are dropped. So $C_{K3;F3} = C_{K3}{}^{\S}_{\S} O_{F3} \vee C_{K3}{}^{\S}_{\S} C_{F3,c7}$. In fact the distinction between these concerns only $c7$, whose precise value is immaterial, so only C_{K3} is of further interest. From $c6 = 1$, we deduce $c5 = 1$. We now decompose $C_{K3} = C_{K4;A1}$ which is just $C_{K4}{}^{\S}_{\S} O_{A1}$ since adders don't fail. Now $c5 = 1$ implies $c2 = 0 \wedge c3 = 1$ or $c2 = 1 \wedge c3 = 0$, giving a disjunction into **TLE.L.L** or **TLE.L.R** for $C_{K4}{}^{\S}_{\S} O_{A1}$.

TLE.L.L: Since $K4 = F1|F2$, we have $C_{K4} = O_{F1}|C_{F2} \vee C_{F1}|O_{F2} \vee C_{F1}|C_{F2}$, with each of C_{F1}, C_{F2} being a disjunction of two faults. However, we earlier derived $c1 = 0$, which is inconsistent with O_{F1} and $J1 = 1$, eliminating a term and forcing $F1.c1$ true. But $c2 = 0$ forces $F1.c2$ true, and we assumed only one fault is ever active in any one component. So we have a contradiction. In such a case we must backtrack to the innermost ancestral nontrivial disjunction, and eliminate the subtree rooted at the relevant disjunct. Thus the subtree at $c2 = 0 \wedge c3 = 1$ is eliminated.

TLE.L.R: As in the previous case we have $F1.c1$ true, but this time $F1.c2$ is false due to $c2 = 1$; so we remain within our constraints. Now $c3 = 0$ forces $F2.c3$ true, and for consistency we must have $F2.c4$ false. This yields a fault configuration for the TLE.

TLE.R: We decompose C_{K2} as in case **TLE.L**, getting $O_{K3}{}^{\S}_{\S} C_{F3} \vee C_{K3}{}^{\S}_{\S} O_{F3} \vee C_{K3}{}^{\S}_{\S} C_{F3}$. The constraint $c1 = 1 \wedge c6 = 0$ and no multiple $F3$ failures, means that this can be made valid by: case **TLE.R.1**, in which $O_{K3}{}^{\S}_{\S} C_{F3,c6}$ holds, with $c5 = 2$; or by case **TLE.R.2**, in which $C_{K3}{}^{\S}_{\S} O_{F3}$ is presumed to hold, with $c5 = 0$; or by case **TLE.R.3**, in which $C_{K3}{}^{\S}_{\S} C_{F3,c6}$ holds, with $c5$ as yet unconstrained; or by case **TLE.R.4**, in which $C_{K3}{}^{\S}_{\S} C_{F3,c7}$ is presumed to hold, with $c5 = 0$.

TLE.R.1: $O_{K3}{}^{\S}_{\S} C_{F3,c6}$ holds, with $c5 = 2$. This is a valid cause of the TLE.

TLE.R.2: We have $C_{K3}{}^{\S}_{\S} O_{F3}$ and $c5 = 0$, so we decompose $C_{K3} = C_{K4;A1} = C_{K4}{}^{\S}_{\S} O_{A1}$ since adders don't fail. Now $c5 = 0$ implies $c2 = c3 = 0$. The latter two imply $F1.c2$ and $F2.c3$ both true, and $c1 = 1$ does not lead to a multiple failure for $F1$. Also $c4 = 1$ is acceptable for $F2$, leading to a valid fault configuration for the TLE.

TLE.R.3: We have $C_{K3}{}^{\S}_{\S} C_{F3,c6}$ as a consequence of which $F3.c6$ holds, and $c5$ is unconstrained. We seek all possible ways of satisfying C_{K3} given the inputs $J1 = 1$ and $J2 = 1$. Now $K3$ is a parallel composition of $F1$ and $F2$, so C_{K3} will contain three terms as usual. Now each of C_{F1} and C_{F2} is a disjunction of two terms, but $c1 = 1$ prevents $F1.c1$ from holding so C_{F1} has just one term that contributes nontrivially. This leads to an overall disjunction of five nontrivial terms.

TLE.R.4: We have $C_{K3}{}^{\S}_{\S} C_{F3,c7}$ and $c5 = 0$. The latter generates only one solution, i.e. $F1.c2$ and $F2.c3$ must both hold.

A tree that depicts the above is shown in Fig. 2. Near the top we show the variable assignments, but suppress them lower down to save space, recording only the fault variables set at various points.[5] Although Fig. 2 is not syntactically a fault tree (FT) according to [30], it is easy to see that it could be straightforwardly transformed into one. We do this in Section 4 after minimisation. At any rate the present tree represents a low level view of the fault analysis.

In terms of the hierarchy of which *Fred* forms a part, a higher level view just represents the fault by a single node: the TLE node itself. Descending the hierarchy thus

[5] The ellipsis in the root indicates that further facts to be accumulated as the analysis descends are to accumulate *inside* the scope of the quantifier (elsewhere, we suppress the ellipsis).

corresponds to growing the more detailed tree along with uncovering the internal structure of *Fred*. Evidently the algorithm described in this section can be easily incorporated into one which deals with large systems in a hierarchical fashion. Whenever a fault tree for a given model has been computed, a pure refinement of a subsystem does not require rebuilding the whole fault tree from scratch. More drastic subsystem evolution, going beyond pure refinements, can imply more widespread changes to fault trees.

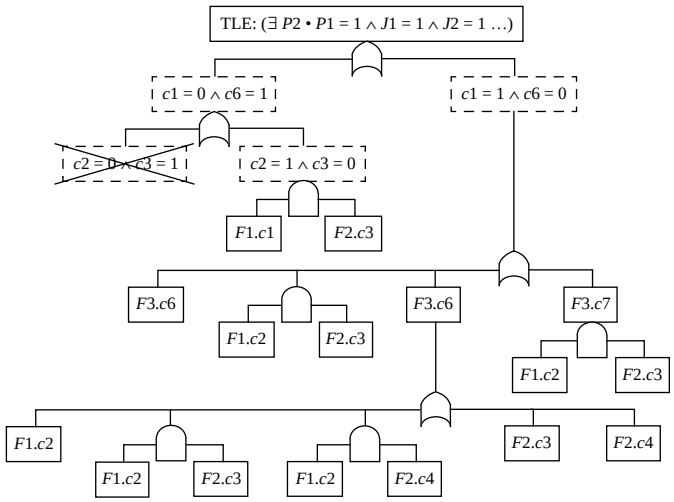

Fig. 2. Part of a Resolution Tree for the TLE of *Fred*.

4 Structured Minimisation

In practical fault analysis we are most interested in minimal fault configurations, the so-called *minimal cut sets* (MCSs for short) consisting of the fewest possible basic faults that cause a particular TLE. The traditional technique for discovering MCSs is subsumption. In principle, one needs to generate all possible configurations that cause a fault, and then check them against one another: any that are subsumed by simpler configurations are discarded. These subsumption checks can be quite expensive for a large system model, since the number of leaves in a tree is exponential in its depth, and the number of subsumption checks is quadratic in the number of leaves. Although in practice efficient algorithms [17, 18, 27, 28] based on binary decision diagrams (BDDs) [16] can be used for this purpose, their worst-case complexity is still exponential in the number of variables of the BDD. In this section we explore ways of reducing the subsumption workload by exploiting the structure of the tree construction as guided by the retrenchment data. The various minimisations are illustrated on the running example.

M.1: Discarding non-needed subtrees. If, during the construction, a fault is generated which leads to an assignment to some variable whose value does not affect the validity of the TLE (eg. there is no dataflow from the fault to the TLE), then the fault node (and, implicitly, any subtree rooted at it) can be discarded immediately since the TLE

is satisfied without it. In general, we call such faults incidental faults. As in the case of the subtree of Fig. 2 rooted at $F3.c7$, which is an example, such faults can arise by considering the disjunction of the complete range of possible faulty configurations of some otherwise needed component.

M.2: Discarding locally subsumed expressions. If, during the construction, a range of options to explore is generated, some of which are subsumed by others, the subsumed options can be discarded immediately. Eg. in Fig. 2, $F2.c3$ subsumes $F1.c2 \wedge F2.c3$. (N.B. The example in **M.1** can also be viewed this way.)

M.3: Discarding subtrees at input-insensitive faults. If, during the construction, a fault is generated which is independent of any input to the component in question, the subtree beneath it can be discarded immediately. Eg. in Fig. 2, $F3.c6$ is a 'stuck_at_0' fault, insensitive to inputs to $F3$. So in **TLE.R**, in considering $O_{K3} \S C_{F3,c6} \vee C_{K3} \S O_{F3} \vee C_{K3} \S C_{F3,c6} \vee \ldots$, the term $C_{K3} \S C_{F3,c6}$ can be discarded immediately in favour of $O_{K3} \S C_{F3,c6}$, even though it is not subsumed by $O_{K3} \S C_{F3,c6}$. (N.B. When $C_{K3} \S C_{F3,c6}$ is eventually decomposed, it *does* yield a family of fault configurations subsumed by $F3.c6$, as is clear from Fig. 2. Such cases can also be viewed as instances of **M.2** *provided* satisfiability of O_{K3} is prima facie unproblematic.)

M.4: Doing final subsumption checking at the subsystem level. The techniques outlined above are not guaranteed to be complete, insofar as further minimisations to generate the MCSs may remain. Rather than leaving these to a final whole-model subsumption check, the brute force subsumption checking to catch them can be done at the subsystem level, since all contributions to the TLE for a fault in a subsystem like *Fred* are causally propagated along data pathways within the subsystem (a structural assumption we take for granted.) Thus the inclusion of the rest of the system will result in an overall description which necessarily factorises, regardless of whether or not the factorisation is obscured (whether to a human observer or to some algorithm) by the complexity of the final expression.

The precise way in which the preceding ideas can be implemented in a tool (such as the FSAP/NuSMV-SA platform [13]) remain a matter for implementation tactics. For example, the subsystem parse could be decorated with suitable directives to prompt the FT generation algorithm to apply certain minimisations when the appropriate point is encountered, or the FT generation algorithm may be written so as to check for the whole range of recognised minimisation opportunities every time another stage in the tree is developed. Internal optimisations, such as the sharing of subcomputations not visible at the FT level, can also be deployed. Details lie beyond the scope of this paper.

When we apply the above to the running example whose resolution tree is indicated in Fig. 2, we get a considerably smaller tree. We transform this into a legal FT as per [30], containing just the MCSs, by accumulating the variable assignments along any path between two logical connectives into the label for an intermediate event (IE), and changing the basic fault nodes into round ones.[6] When we do all this, we end up with the minimised fault tree in Fig. 3.

[6] N.B. Where a basic fault occurs in the interior of the resolution tree (eg. the subtrees at $F3.c6$ or $F3.c7$ in Fig. 2, were these trees not discarded), the subtree is manipulated to distribute the interior basic fault into the nearest descendant conjunction(s), and IEs are generated labelled by the relevant logical combinations of the IEs at the roots of the subtrees thus affected.

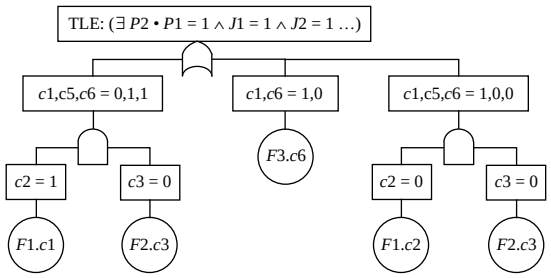

Fig. 3. A Minimised Fault Tree for the TLE of *Fred*.

5 Timing and Internal State

Up to now everything has been treated as instantaneous, and the job of fault tree analysis has simply been to trace the possible functional (or more generally, relational) dependencies that connect the inputs and outputs in a given TLE, and thereby, to display the connections between the primitive faults that contribute to valid instances of the TLE. This instantaneous assumption is obviously not adequate for all situations of interest, and so in this section, we introduce a model of time, in order to capture the behaviour of systems in which time delays cannot be neglected. Since many of the digital components that are found in circuits such as our running example are stateful, this generalisation is an important one.

We introduce discrete time, with ticks labelled by integers, and for ease of exposition we just modify slightly our running example. The adders will remain stateless, delivering their result instantaneously, while the fanouts will introduce a unit delay between an input received and the outputs delivered. Thus while the definition of the adders remains unaltered aside from the introduction of a time parameter, eg. $A1_A(\langle a2(t), a3(t)\rangle, a5(t)) \equiv a5(t) = a2(t) + a3(t)$, the definition of the fanouts becomes eg. $F1_A(I1(t), \langle a1(t+1), a2(t+1)\rangle) \equiv a1(t+1) = a2(t+1) = I1(t)$. As well as this, the fault variables become time dependent (to permit the description of eg. glitches), but otherwise, the relational descriptions of components are time independent. So the faulty behaviour of $F1$ becomes:

$$
\begin{array}{l}
F1_C(J1(t), \langle c1(t+1), c2(t+1)\rangle) \equiv \\
\quad (F1.c1(t+1) \Rightarrow c1(t+1) = 0) \wedge (F1.c2(t+1) \Rightarrow c2(t+1) = 0) \wedge \\
\quad \neg(F1.c1(t+1) \wedge (F1.c2(t+1)) \qquad \textit{ELSE\_JDEAL}
\end{array}
$$

Let *Fred* with these alterations be renamed *FreT* (we will continue to refer to Fig. 2). With this change, the retrenchment data for *FreT* become:

$$
\begin{array}{l}
W_{F1}(I1(t), J1(t)) \equiv \mathsf{true} \\
O_{F1}(\langle a1(t+1), a2(t+1)\rangle, \langle c1(t+1), c2(t+1)\rangle, I1(t), J1(t)) \equiv \\
\qquad (c1(t+1) = c2(t+1) = J1(t)) \\
C_{F1}(\langle a1(t+1), a2(t+1)\rangle, \langle c1(t+1), c2(t+1)\rangle, I1(t), J1(t)) \equiv \\
\qquad (F1.c1.0(t+1) \wedge c1(t+1) = 0 \wedge c2(t+1) = J1(t)) \vee \\
\qquad (F1.c2.0(t+1) \wedge c1(t+1) = J1(t) \wedge c2(t+1) = 0)
\end{array}
$$

(We omit the retrenchment data for the adders, which just get labelled by '(t)'.)

With this setup, in fact very little changes as regards the top down resolution driven fault tree analysis, *provided* we remember that our subsystems are all still finite component, finite signal, finite state, and acyclic. The reason that there is little change is that the set of paths through the subsystem, between inputs and outputs, remains unaltered by the mere introduction of time delays along them – fault tree analysis (in the sense of this paper) can in the end be seen as a deductive process about such paths and sets of such paths. The fault trees resulting from the time sensitive analysis can of course be differently shaped from those in the time independent one, since the same component may contribute in different ways at different times.

To illustrate the above, let us do an analysis for the *FreT* subsystem of the same TLE we considered previously, but this time with the output $P1$ instantiated to 1 for some time t (and otherwise unspecified), and with inputs held constant at 1 as before, which we write as $J1 = J2 = P1(t) = 1$. Doing the analysis as described in Section 3, but this time noting the time labels along the way, and then doing the minimisation as described in Section 4, we get the FT in Fig. 4, in which preprimes denote the value at $t-1$, and the labelling of the IEs is incomplete for space reasons (a full labelling would cite values at t and at $t-1$ for several variables). Note how the fact that the output is not required to be constant, has spawned a valid instance of the branch of the FT that was cut off in Fig. 2. We are only demanding a glitch, so the two $F1$ faults that could not coexist statically, are permitted to occur at successive instants. Of course if we asked for the glitch to persist for two time ticks, this branch would get cut off once more. This example vividly illustrates the increased expressive power gained by adding timing to essentially the same techniques that we discussed statically.

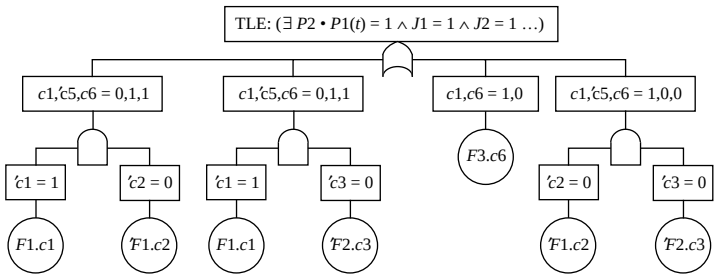

Fig. 4. A Minimised Fault Tree for the TLE of *FreT*.

6 Introducing Feedback

The (technically) relatively mild generalisation of the last section becomes more interesting when we include feedback as well as timing delays. We modify our subsystem *FreT* by removing $A3$ and $F2$, and introducing a feedback signal (called k in the concrete system) from $F3$ to $A1$, resulting in subsystem *Jim*. See Fig. 5.

Now we can no longer rely on a static syntactic description of the subsystem as the analysis proceeds, but must unfold a recursive structure. The essentials of this are:

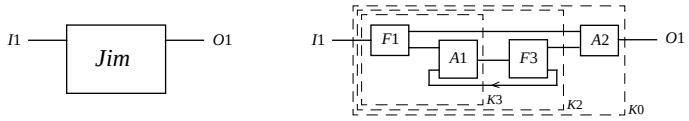

Fig. 5. A subsystem *Jim* with cyclic internal structure.

$$O1(t) = (c1(t) + c6(t)) \quad ; \quad c1(t) = c2(t) = I1(t-1)$$
$$c6(t) = c5(t-1) = (c2(t-1) + k(t-1)) = (c2(t-1) + c5(t-2))$$

This is a standard feedback control system, and its I/O behaviour can be computed by standard means. Performing the required back substitutions (details omitted for lack of space), we get:

$$O1(t) = I1(t-1) + I1(t-2) + I1(t-3) + \ldots \; = \; \sum_{q=1}^{\infty} I1(t-q)$$

This is a shorthand for describing an infinite set of possible finite behaviours, on the understanding that all values are (bounded) natural numbers, and that an at most finite number of the $I1$ values in the summation are non-zero (and that one cuts off the summation at some point after the earliest non-zero value, to represent initialisation at a point in the finite past). The proliferation in behaviours is due to the fact that $A1$'s output remains stable when its $c2$ input is 0, so that the value held in $F3$ (and hence output at $O1$) remains invariant as long as $I1$ continues to remain at 0. Thus if we stipulate $O1(t) = 2$, then this can arise via $I1(t-1) = 2$, or via $I1(t-2) = 2$, or $I1(t-3) = 2$, etc. (with all other $I1$ values zero). Alternatively we could have $I1(t-1) = 1 \wedge I1(t-2) = 1$, or $I1(t-1) = 1 \wedge I1(t-3) = 1$, or $I1(t-1) = 1 \wedge I1(t-4) = 1$, or …etc., or $I1(t-2) = 1 \wedge I1(t-3) = 1$, or $I1(t-2) = 1 \wedge I1(t-4) = 1$, etc. etc.

Admittedly we have been considering the fault-free behaviour of *Jim* for the sake of simplicity, but there is no reason at all why similar situations should not arise during the analysis of genuine faults. The back substitutions performed from the TLE '$O1(t) = 2$' are exactly the steps that a retrenchment based fault analysis would dictate.

There are at least three approaches to the question of there being an infinity of possible causes of some situation, just raised. Firstly one could simply regard the TLE as underspecified, since it places no constraints on the input values. Any finite constraint on these that is consistent with the TLE and supplies all the values 'needed' by the TLE[7] would immediately reduce the set of possible causes to a finite one, eliminating the problem. Essentially we would be placing an a priori bound on how far in the past the earliest of the causes of the TLE had occurred, an approach that in general is incomplete.

Secondly, one could examine what a standard model checking approach (such as the FSAP/NuSMV-SA platform [13]) would deliver. Such approaches work by exhaustive search of the state space of the system, keeping an eye out for states already encountered along a given path. Finding a repeated occurrence of the same state, cuts off the search, since it is interpreted as looping behaviour in the system. In our example this would yield a finite representation of the infinite set of possible behaviours, analogously to the way that the infinite summation above is a finite representation of it.

[7] We are being rather imprecise here about the definition of neededness, since it would depend on the precise nature of the components in the subsystem and their interdependencies.

The stable behaviour of the adders previously alluded to is reflected in self-loops on the relevant states generated by the state space search algorithm. Such a representation would require some interpretation as regards the generation of fault trees, since a naive FT generation algorithm would attempt to generate a tree with infinite disjunctions, and not terminate. Adding a finite starting point in the past is an easy way to prevent this, although as above, an a priori finite bound in the past makes the approach incomplete.

A third, and most sophisticated approach to the issue, is to honestly take on board the control nature of the cyclic system, and to combine the model checking strategy with deeper insights about control systems.[8] The benefits of such an approach are that it could yield a complete description, by representing recursive parts of the set of behaviours in a suitably symbolic manner, even extending to situations in which the state space is not finite. However all of this would require deep insight into the relationship between decidabilities in the relevant model checking and control theory domains, since it is well known that combining theories which are decidable on their own, does not automatically lead to decidability of the combination.

7 Conclusions

In this paper we have presented a formal account of fault tree generation based on retrenchment. We have shown how the retrenchment framework is able to capture several aspects of the fault tree generation, namely the mechanical construction of a fault tree based on structural information, fault tree minimization, system model evolution based on a hierarchy of models viewed at multiple levels of abstraction, and fault injection. Finally, the approach can be generalised to deal with dynamic and cyclic systems.

Our work has been inspired by Hip-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [24, 25, 26], a framework incorporating a mechanical fault tree synthesis algorithm based on system structure, and taking into account model evolution. The synthesis of the fault tree is based on a preliminary functional failure analysis (FFA) and a tabular technique (IF-FMEA) used to generate a model of the local failure behaviour, activities normally performed manually during system design and safety assessment. Our work addresses the automation of the whole process assuming that a formal specification of both system and fault model is available. Furthermore, we have shown how the synthesis algorithm can be coupled with suitable tactics to perform local minimal cut-set computation, reducing the overall computational effort.

Our techniques can be incorporated into formal tools supporting the safety assessment of complex systems, like the FSAP/NuSMV-SA platform [13, 20]. The algorithms described here improve over the ones used there for two reasons. First, they allow the generation of structured fault trees, which are more informative than the flat fault trees produced by the current FSAP platform. Second, they allow the taking of dynamic information into account, eg. they can deal with transient failures (Section 5), and feedback (Section 6). While our focus was on automatic synthesis, the DIFTree (Dynamic Innovative Fault Tree) [22] methodology, implemented in the Galileo tool [29], is mainly

[8] Certainly the calculation indicated above is standard feedback control theory, and such calculations have been automated (in a suitably symbolic manner) in standard control theory toolkits eg. SIMULINK [1, 21].

concerned with the problem of fault tree evaluation. It uses a modularisation technique [19] to identify (in linear time) independent sub-trees, that can be evaluated using the most appropriate techniques (BDD-based techniques for static fault trees, Markov techniques or Monte Carlo simulation for dynamic ones). In addition, it supports different probability distributions for component failures. A similar modularisation and decomposition technique is advocated in [2]. That technique is orthogonal to our notion of structural generation; in particular, it is concerned with isolating different sub-trees that can be synthesised (or evaluated) separately, whereas our structural information can be used to synthesise (or evaluate) each sub-tree on its own.

Although an experimental evaluation of our algorithm was beyond the scope of this paper, we have provided many hints about the advantages such an algorithm would have with respect to the traditional monolithic algorithms which just flatten the model. First, it makes it possible to synthesise the fault tree by considering each component in isolation, thus avoiding building an internal representation of the whole model (eg. avoiding the generation of a BDD for it). Second, it suggests that the MCS computation can benefit from local minimisation. As future work, we wish to design a practical implementation and evaluate it experimentally against state-of-the-art techniques, eg. the BDD-based routines [17, 18, 27, 28] used in the FSAP platform. Given that integer constraint solving is needed to deal with time, we foresee that there might also be room for using decision procedures for such a theory, eg. MathSAT [11, 23]. Finally, we would like to integrate such algorithms into the FSAP platform [13].

Further issues we would like to address include dynamic aspects (see eg. [22]), that we have only sketched in this paper for lack of space. In particular, we would like to investigate the problem of sequential dependencies and failure duration, and their representation inside the fault tree. Finally, it would be interesting to adapt our algorithms to the truncated computation of prime implicants described in [28].

References

[1] Tewari. A. *Modern Control Design With MATLAB and SIMULINK*. Wiley, 2002.

[2] A. Anand and A.K. Somani. Hierarchical Analysis of Fault Trees with Dependencies, using Decomposition. In *Proc. Annual Reliability and Maintainability Symposium*, pages 69–75, 1998.

[3] R. Banach. Retrenchment and system properties. Submitted.

[4] R. Banach and R. Cross. Safety requirements and fault trees using retrenchment. *Proc. SAFECOMP-04, Springer*, Heisel, Liggesmeyer, Wittmann (eds.), LNCS Volume 3219:210–223, 2004.

[5] R. Banach and C. Jeske. Output retrenchments, defaults, stronger compositions, feature engineering. Submitted.

[6] R. Banach and M. Poppleton. Engineering and theoretical underpinnings of retrenchment. Submitted.

[7] R. Banach and M. Poppleton. Retrenchment: An engineering variation on refi nement. *B'98: Recent Advances in the Development and Use of the B Method: Second International B Conference, Montpellier, France, LNCS*, 1393:129–147, 1998.

[8] R. Banach and M. Poppleton. Retrenchment and punctured simulation. *Proc. IFM-99, Springer*, Araki, Gallway, Taguchi (eds.):457–476, 1999.

[9] R. Banach and M. Poppleton. Sharp retrenchment, modulated refi nement and punctured simulation. *Form. Asp. Comp.*, 11:498–540, 1999.

[10] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Requirements Engineering Journal*, 8:266–288, 2003.
[11] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. Mathsat: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning, Special Issue on SAT*, 2006. To appear.
[12] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving safety assessment of complex systems: An industrial case study. *International Symposium of Formal Methods Europe (FME 2003), Pisa, Italy, LNCS*, 2805:208–222, September 2003.
[13] M. Bozzano and A. Villafiorita. Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. *Computer Safety, Reliability, and Security, LNCS*, 2788:49–62, 2003.
[14] M. Bozzano and A. Villafiorita. Integrating fault tree analysis with event ordering information. *Proc. ESREL 2003*, pages 247–254, 2003.
[15] M. Bozzano, A. Villafiorita, et al. ESACS: An integrated methodology for design and safety analysis of complex systems. *Proc. ESREL 2003*, pages 237–245, 2003.
[16] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
[17] O. Coudert and J.C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *Proc. Design Automation Conference (DAC 1992)*, pages 36–39. IEEE Computer Society Press, 1992.
[18] O. Coudert and J.C. Madre. Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In *Proc. Annual Reliability and Maintainability Symposium (RAMS 1993)*, 1993.
[19] Y. Dutuit and A. Rauzy. A Linear-time algorithm to find modules in fault trees. *IEEE Transactions on Reliability*, 45(3):422–425, 1996.
[20] The FSAP/NuSMV-SA platform. `http://sra.itc.it/tools/FSAP`.
[21] Nuruzzaman. M. *Modeling And Simulation In SIMULINK For Engineers And Scientists*. Authorhouse, 2005.
[22] 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. High-Assurance Systems Engineering Symposium (HASE 1998)*, pages 21–28. IEEE, 1998.
[23] MathSAT. `http://mathsat.itc.it`.
[24] Y. Papadopoulos. *Safety-directed system monitoring using safety cases*. PhD thesis, Department of Computer Science, University of York, 2000. Tech. Rep. YCST-2000-08.
[25] Y. Papadopoulos and M. Maruhn. Model-Based Synthesis of Fault Trees from Matlab-Simulink Models. In *Proc. Conference on Dependable Systems and Networks (DSN 2001)*, pages 77–82, 2001.
[26] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner. Analysis and Synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71(3):229–247, 2001.
[27] A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering and System Safety*, 40(3):203–211, 1993.
[28] 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.
[29] K.J. Sullivan, J.B. Dugan, and D. Coppit. The Galileo Fault Tree Analysis Tool. In *Proc. Symposium on Fault-Tolerant Computing (FTCS 1999)*, pages 232–235. IEEE, 1999.
[30] 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.