Under consideration for publication in Formal Aspects of Computing

The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment I: Combinational Circuits [1]

Richard Banach[1] and Marco Bozzano[2]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
banach@cs.man.ac.uk,
[2]FBK-IRST, Via Sommarive 18,
Povo, 38123 Trento, Italy
bozzano@fbk.eu

Abstract. The manual construction of fault trees for complex systems is an error-prone and time-consuming activity, 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. 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.

Keywords: Fault Tree Analysis, Fault Injection, Retrenchment, Mechanical Fault Tree Synthesis, Combinational Circuits

1. Introduction

Safety 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. In aeronautics, for instance, safety requirements stating the (degraded) conditions under which systems must remain operational are defined along with the other system requirements. Safety analysis aims to identify all possible hazards of the system, in order to ensure that the system meets the safety requirements demanded. Safety analysis is particularly critical in the case of reactive systems, that is systems with infinite time behaviour, because hazards can be the result of complex interactions involving the dynamics of the system [Siu94].

Correspondence and offprint requests to: Richard Banach, School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, U.K. email: banach@cs.man.ac.uk

[1] Work partly supported by the E.U. projects ISAAC, contract no. AST3-CT-2003-501848, and MISSA, contract no. ACP7-GA-2008-212088.

One popular safety analysis method is Fault Tree Analysis (FTA) [VGRH81, VSD+02]. It is a deductive analysis, which, given the specification of an undesired state (e.g. a failure state), usually referred to as *top level event* (TLE), systematically builds all possible chains of one of more basic faults that contribute to the occurrence of the event. Fault trees provide a convenient symbolic representation of the combination of events causing the TLE, and they are usually represented as a parallel or sequential combination of logical gates. The problem of fault tree generation for reactive systems is substantially harder than in the traditional combinational case, due to the presence of dynamics, which can influence the presence of failures.

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; moreover, managing the generated fault trees is challenging in case of large complex systems. Therefore, in recent years there has been a growing interest in model-based techniques and tools to automate the production of fault trees [BV+03b, B+06], such as the FSAP safety analysis platform [BV07, FSA].

The starting point for this paper is our previous work relating retrenchment [Ret, BPJS07, BJP08] and formal system model evolution [BC04]. In [BC04] we showed how retrenchment, as opposed to conventional refinement, can provide a formal account of the so-called fault injection process [BV07], that takes the abstract system model (that is the model of the system in nominal conditions), to 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 and a companion paper, which we refer to below as PaperII [BB10], we show how retrenchment can be developed into a versatile structured approach for the mechanical construction of fault trees for reactive systems.[2] Building on the ideas sketched in [BC04], 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.

In the present paper we develop these techniques for the simpler case of combinational circuits, extending to timed acyclic circuits, and timed circuits with feedback in PaperII. This paves the way for the automated safety analysis of reactive systems. The techniques we propose usefully complement more traditional safety assessment techniques, in particular those based on fault tree analysis. Our techniques rely on a formal model of the design and of faulty behavior, whereas traditional techniques rely on human expertise and knowledge of the system at hand. Hence, a manual review of the automatically generated fault trees may help the design and safety engineers identify errors (or limitations) in design and fault models, or in manually produced results, and possibly suggest either changes in the model themselves, or system redesign recommendations.

The techniques we present in this paper and in PaperII improve over the ones discussed in [BV07], 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 [BV07], thus facilitating manual review by safety analysts. Furthermore, we exemplify how these techniques can be adapted to generating the minimal cut sets of a fault tree. We show that, by processing the generated subtrees on the fly, 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 set up our basic concepts for systems and their compositions, and present retrenchment and relevant notions of composition for retrenchments. In Section 3 we give an overview of fault tree analysis and discuss the role of fault trees in the traditional safety and reliability engineering process. In Section 4 we present our retrenchment directed approach to the generation of structured fault trees on a running example. The technique is based on resolution, and the resolution trees generated by the underlying algorithm can then be post-processed into fault trees having a standard format. Section 5 formalises the approach, showing, crucially, that the generation of resolution trees is sound and complete. The trees thus generated can be used for various purposes, and we cover their post-processing into fault tree form. In Section 6 we show how the structured analysis can be modified to reduce the work of finding the minimal cut sets of some fault condition. Section 7 formalises these insights, making them rigorous in the context of Section 5. This completes our retrenchment based treatment of fault tree generation for combinational circuits. Section 8 discusses related work, and Section 9 concludes, looking forward to PaperII.

Notation: In the rest of the paper we use relational techniques. Invariably, we refer to a relation by talking about the predicate which defines it.

[2] Our techniques generate so-called *resolution trees*, which can be mechanically transformed into traditional fault trees, whenever needed.

2. Systems, Retrenchments and Compositions

In this paper we approach the mechanisable construction of fault trees for system models via the technique of fault injection. Fault injection changes the fault-free system model into the faulty system model, by making alterations to the definition of its behaviour that usually make the two system models incompatible with each other. Relating two incompatible models of this kind is not trivial theoretically: most theoretical approaches are concerned with progressing a system abstraction towards code, namely with *refinement*. Fault injection is therefore best viewed as a special kind of *system evolution*. For this we need a formal approach to system evolution, namely retrenchment [Ret, BPJS07, BP98, BP03, BJP08] — it proves flexible enough to capture the kind of system changes we have in mind. This section is concerned with developing the technical tools we need.

We focus on synchronous hardware systems, and we describe our models using collections of input/output transformers, one transformer per component or collection of components, depending on the granularity of the description. An I/O transformer is a relation from the system's inputs to the system's outputs. For the combinational circuits of this paper, the I/O transformers act instantaneously.

2.1. Circuits and their Compositions

Combinational digital hardware circuits typically have some input signals and some output signals. Henceforth we will assume that all circuits are **acyclic**, and also **time invariant** i.e. that observed behaviour does not depend on time. We will also assume that circuits are **total** as relations from inputs to outputs, i.e. in I/O terminology, they will be **input ready**. This will be useful later, and is inevitably the case in practice. For convenience, we also record here that all data types we use are **finite**, as one would expect in a digital hardware context.

Let *AcyOp* (short for *AcyclicOperation*) be such a circuit, and suppose that *AcyOp* has three inputs i_0, i_1, i_2, and three outputs o_0, o_1, o_2. Then we can represent the behaviour of *AcyOp* via the relation $AcyOp(\langle i_0, i_1, i_2\rangle, \langle o_0, o_1, o_2\rangle)$ where we have grouped the inputs and outputs using angle brackets. We next consider composition mechanisms for such relations.

2.1.1. Parallel Composition:

If $AcyOp_1$ and $AcyOp_2$ are two I/O transformers, with the same signatures as *AcyOp* for convenience, but with additional subscripts 1 and 2 respectively, then their parallel composition is given by:

$$\begin{aligned} &AcyOp_{1|2}(\langle i_{10}, i_{11}, i_{12}, i_{20}, i_{21}, i_{22}\rangle, \langle o_{10}, o_{11}, o_{12}, o_{20}, o_{21}, o_{22}\rangle) \equiv \\ &\quad AcyOp_1(\langle i_{10}, i_{11}, i_{12}\rangle, \langle o_{10}, o_{11}, o_{12}\rangle) \wedge AcyOp_2(\langle i_{20}, i_{21}, i_{22}\rangle, \langle o_{20}, o_{21}, o_{22}\rangle) \end{aligned} \tag{1}$$

2.1.2. Skew-Sequential Composition:

Suppose that $AcyOp_1$ and $AcyOp_2$ have the signatures and dependency structure of *AcyOp*, and suppose we want to compose them sequentially by identifying signals of $AcyOp_1$ and $AcyOp_2$ as follows:

$$o_{10} = i_{21} \qquad\qquad o_{11} = i_{22} \tag{2}$$

We note that to avoid ambiguity, we need to specify which signals are to be identified. The general case, in which it is not necessary that *all* the output signals of $AcyOp_1$ are connected to *all* the input signals of $AcyOp_2$, explains why it is called the *skew*-sequential composition. Let us write the signal identifications in (2) as δ. This will give us $AcyOp_{1;_\delta 2}$. We suppress the '$_\delta$' subscript when the connecting signals can be understood from the context. In our example, o_{12} becomes an output signal of the composed circuit. Note that if there are *no* identified signals, skew-sequential composition reduces to parallel composition.

The composed I/O relation $AcyOp_{1;_\delta 2}$ can be calculated by composing the I/O relations $AcyOp_1$ and $AcyOp_2$, along the connecting signals. Fig. 1 gives an idea of what this produces. We thus obtain the following:

$$\begin{aligned} &AcyOp_{1;_\delta 2}(\langle i_{10}, i_{11}, i_{12}, i_{20}\rangle, \langle o_{12}, o_{20}, o_{21}, o_{22}\rangle) \equiv \\ &\quad (\exists x, y \bullet AcyOp_1(\langle i_{10}, i_{11}, i_{12}\rangle, \langle x, y, o_{12}\rangle) \wedge AcyOp_2(\langle i_{20}, x, y\rangle, \langle o_{20}, o_{21}, o_{22}\rangle) \end{aligned} \tag{3}$$

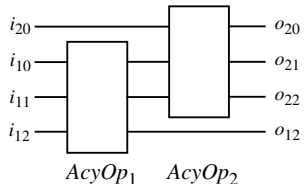

Fig. 1. The skew-sequential composition of circuits $AcyOp_1$ and $AcyOp_2$.

2.2. Dataflow Retrenchment

We now turn to retrenchment. Retrenchment was designed for evolution steps between system models described via general state transition systems. As such, it deals with transitions that feature not only the consumption of an input i and the generation of an output o, but also a change of the internal state of the system. This full generality will be needed in PaperII, which deals with stateful systems, but here, to deal with purely combinational digital circuits, we can restrict to a simpler state-free formulation: dataflow retrenchment. In particular, no separate initialisation criteria are needed, as they would be if there were initial states to worry about.

Retrenchment relates two system models, an abstract one *Abs*, and a concrete one *Conc*, say. We will assume that the transitions of *Abs* collect into *operations*, e.g. $Op_A(i,o)$, which are relations (I/O transformers) of the kind discussed above. For *Conc*, the corresponding operation will be $Op_C(j,p)$, so that j represents its input(s), and p its output(s).

In this simple dataflow formulation, a retrenchment from *Abs* to *Conc* consists of two things. Firstly, for every corresponding pair of operations, $Op_A(i,o)$ and $Op_C(j,p)$, there are three relations, the *within* relation $W_{Op}(i,j)$, the *output* relation $O_{Op}(o,p,i,j)$, and the *concedes* relation $C_{Op}(o,p,i,j)$ — collectively these are called the retrenchment data. Secondly (and on a per-operation basis), the operations and retrenchment data must satisfy the correctness proof obligation (PO):

$$W_{Op}(i,j) \wedge Op_C(j,p) \Rightarrow (\exists o \bullet Op_A(i,o) \wedge (O_{Op}(o,p,i,j) \vee C_{Op}(o,p,i,j))) \tag{4}$$

This says that provided the within relation is true for a pair of abstract and concrete inputs i,j, whenever there is a transition of the concrete operation (from j), then there is a transition of the corresponding abstract operation (from i), such that the resulting abstract and concrete outputs o,p *either* establish the output relation $O_{Op}(o,p,i,j)$, *or* they establish the concedes relation $C_{Op}(o,p,i,j)$. Each choice of different retrenchment data for *Abs* and *Conc* yields a different retrenchment, and makes a different statement about the relationship between them.

The absence of state from the simple dataflow formulation of retrenchment just given, brings it much closer to conventional notions of refinement for pure I/O systems than would be the case if state were present — the great deviation from this being, of course, the presence of the concession C_{Op} in the conclusion of the PO.

Thus, the within relation (for a given operation) acts just like an input transformer, though being an *a priori* unrestricted relation (to aid modelling in situations where the relationship between abstract and concrete systems is not as close as one would mathematically like), it is not subject to the totality or surjectivity restrictions that one often finds in refinement formulations. We can view the within relation as restricting the scope of the statement about the relationship between *Abs* and *Conc* that one wishes to make (due, perhaps, to the intrinsic incompatibility between them in certain areas), allowing us to make more incisive statements about those parts of the two systems that *are* more compatible. Note that due to the 'don't care' interpretation of the implication connective '$\Rightarrow$' when the hypotheses of the PO (4) are not all true, it is vital that the retrenchment user assures himself that the within relation employed in any given situation covers a wide enough range of circumstances — i.e. *everything* that one might wish to be covered by the retrenchment analysis, taking all other considerations into account.[3]

In the dataflow retrenchment formulation, the other two relations (for a given operation), give the capability of partitioning the behaviour of the related abstract and concrete pair of operations into two parts. Aside from their names, O_{Op} and C_{Op}, there is nothing to distinguish them, since they both feature the same set of variables. However,

[3] To give a small informal example of the flexibility of the within and concedes relations, consider an arithmetic operation on bounded fixpoint values that generates some exceptional situation if the result overflows. In the general case, the within relation would be relatively weak, permitting inputs that caused the exception, and needing a nontrivial concedes relation to handle the exceptional circumstances. On the other hand, *if the context offered a guarantee that exceptional circumstances would not arise*, the within relation could be strengthened to restrict consideration to those inputs that were possible, and the concedes relation could be trivialised, since the context guarantee ensured that it was not needed.

if state were present (as it will be in PaperII), then O_{Op} would be coupled to a further relation which is absent from the dataflow formulation, the retrieve relation, which relates abstract and concrete states, and O_{Op} would thereby be much more strongly associated with 'good' behaviour, allowing C_{Op} to cover 'bad' behaviour. Accordingly, we will later use O_{Op} to describe nominal behaviour, and will use C_{Op} to cover faulty behaviour.

For the work of the rest of the paper it is not the PO itself that will be used, but the associated simulation relation, the statement that all the conjuncts of both the antecedent and consequent of the PO are true:

$$\Sigma^1 \equiv W_{Op}(i,j) \wedge Op_A(i,o) \wedge Op_C(j,p) \wedge (O_{Op}(o,p,i,j) \vee C_{Op}(o,p,i,j)) \tag{5}$$

2.3. Compositions of Retrenchments

Now we extend our results on parallel and skew-sequential compositions of I/O relations for circuits, to parallel and skew-sequential compositions of retrenchment data concerning abstract and concrete versions of them.

To minimise the proliferation of cumbersome formulae, we reuse notations established earlier, and so both compositions will be based on our running example *AcyOp*. To construct compositions of retrenchments, we need four versions of *AcyOp*, namely: $AcyOp_{1,A}, AcyOp_{2,A}, AcyOp_{1,C}, AcyOp_{2,C}$. Of these, the *A*-subscripted abstract ones $AcyOp_{1,A}$, $AcyOp_{2,A}$, have the variables that appear in (1) for $AcyOp_1$ and $AcyOp_2$ respectively; and the *C*-subscripted concrete ones $AcyOp_{1,C}, AcyOp_{2,C}$, have similar variables, but with all occurrences of input variables *i* replaced by *j*, and all occurrences of output variables *o* replaced by *p*.

We also need retrenchment data. Thus the retrenchment from $AcyOp_{1,A}$ to $AcyOp_{1,C}$ will have data: $W_{AcyOp,1}$, $O_{AcyOp,1}$, $C_{AcyOp,1}$, and the retrenchment from $AcyOp_{2,A}$ to $AcyOp_{2,C}$ will have data: $W_{AcyOp,2}, O_{AcyOp,2}, C_{AcyOp,2}$.

Both parallel and skew-sequential composition of retrenchments can easily be shown to be sound, in the sense that if the retrenchment data and operations of each component satisfy the correctness PO individually, then the combined operations satisfy the correctness PO using the combined retrenchment data. We do not cover these soundness arguments here, see [BJP08]. The soundness arguments extend readily to the simulation relations: if the simulation relations for the components are valid, then the simulation relations for the combinations are valid too.

2.3.1. Parallel Composition:

Given the notations just described, the construction of the parallel composition of two retrenchments, from Abs_1 to $Conc_1$, and Abs_2 to $Conc_2$ is straightforward. We use $|$ and $\vee$ for the parallel composition and union of independent relations (which correspond to $\wedge$ and $\vee$ for the predicates which denote those relations). The parallel composition of the operations is as in (1):

$$AcyOp_{1|2,A} \equiv AcyOp_{1,A} \,|\, AcyOp_{2,A} \tag{6}$$

$$AcyOp_{1|2,C} \equiv AcyOp_{1,C} \,|\, AcyOp_{2,C} \tag{7}$$

In this, $AcyOp_{1|2,A}$ is just like (1), except that the identifiers $AcyOp_{1|2,A}, AcyOp_{1,A}, AcyOp_{2,A}$ in (6) and (7) contain an extra '*A*' subscript. The story for $AcyOp_{1|2,C}$ is similar except that the extra subscript is '*C*', and all *i* and *o* variables become *j* and *p* variables respectively.

With the operations defined, the retrenchment data for the parallel composition are given in terms of the component retrenchment data as follows:

$$W_{AcyOp,1|2} \equiv W_{AcyOp,1} \,|\, W_{AcyOp,2} \tag{8}$$

$$O_{AcyOp,1|2} \equiv O_{AcyOp,1} \,|\, O_{AcyOp,2} \tag{9}$$

$$C_{AcyOp,1|2} \equiv O_{AcyOp,1} \,|\, C_{AcyOp,2} \vee C_{AcyOp,1} \,|\, O_{AcyOp,2} \vee C_{AcyOp,1} \,|\, C_{AcyOp,2} \tag{10}$$

What these formulae say is actually rather simple. The combined within relation is just the combination of the two withins, each acting on its own variables. The combined output relation is similar. The combined concedes relation expresses a disjunction of three possibilities: either the first system is well behaved and establishes the output relation while the second fails and establishes the concession, or *vice versa*, or both fail and establish their concessions.

2.3.2. Skew-Sequential Composition:

Skew-sequential composition is built on similar lines. We reuse the notation established above, and sequentially compose both abstract and concrete circuits by identifying output signals of $AcyOp_{1,A}$ and $AcyOp_{1,C}$ with input signals of

$AcyOp_{2,A}$ and $AcyOp_{2,C}$ respectively via an extension of equations (2):

$$\begin{array}{llll} o_{10} = i_{21} & & o_{11} = i_{22} & \\ p_{10} = j_{21} & & p_{11} = j_{22} & \end{array} \tag{11}$$

Let us write these signal identifications as δ, and the sequential composition of relations based on these identifications as ${}^{\circ}_{\circ\delta}$. The skew-sequentially composed I/O relations that arise are simple syntactic modifications of (3). Thus, on the abstract side, we have (3) precisely, except that the various relation names have an additional A subscript. On the concrete side, the subscript is C, but also, the various input variables are all called j instead of i (although they carry the same subscripts as in (3)), and the output variables are called p instead of o (again carrying the same subscripts).

The most complicated piece of the composed retrenchment data for a skew-sequential composition is the within relation. We give the derivation next, together with a discussion of a number of assumptions made on the way, but it will turn out below that we need only a trivial case of it for our application, namely the relation given by true. Readers can skip over the derivation to (16) if they wish.

To make the skew-sequential composition of dataflow retrenchments for our circuits go smoothly, we henceforth assume that the within relations for a circuit decompose into a conjunction of individual signal-pair subrelations thus:

$$W_{AcyOp,1}(\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \equiv W_{AcyOp\cdot 0,1}(i_{10},j_{10}) \wedge W_{AcyOp\cdot 1,1}(i_{11},j_{11}) \wedge W_{AcyOp\cdot 2,1}(i_{12},j_{12}) \tag{12}$$

$$W_{AcyOp,2}(\langle i_{20},i_{21},i_{22}\rangle,\langle j_{20},j_{21},j_{22}\rangle) \equiv W_{AcyOp\cdot 0,2}(i_{20},j_{20}) \wedge W_{AcyOp\cdot 1,2}(i_{21},j_{21}) \wedge W_{AcyOp\cdot 2,2}(i_{22},j_{22}) \tag{13}$$

The representation in (12) and (13) embodies quite a strong assumption on the way that signals behave. What justification is there for making it? We argue, for a contradiction, that if the signals in the systems that we represent via our models do *not* behave in a way that allows us to decouple their behaviour in the way suggested by (12) and (13), then they are not behaving as 'pure wires', which simply transmit what they receive to their output. On the contrary, they must be viewed as complex devices in their own right, to be modelled by more complex multivariable relations in order to express the crosstalk that prevents the decoupled treatment of (12) and (13). This also ties in with the skew-idea, in that the incomplete matching of the outputs of the first layer to the inputs of the second layer is conceptually problematic if there is crosstalk between signals some of which are connected to a further component and some of which are not. We are forced to demand decoupling at least between those signals which are connected to the further component and those that are not, but the even simpler picture of (12) and (13) in fact proves to be sufficient.

With the above understood, we can write down the composed within relation, the most complicated piece of the composed retrenchment data:

$$\begin{array}{l} W_{AcyOp,1;_{\delta}^{\circ}2}(\langle i_{10},i_{11},i_{12},i_{20}\rangle,\langle j_{10},j_{11},j_{12},j_{20}\rangle) \equiv W_{AcyOp,1}(\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \wedge W_{AcyOp\cdot 0,2}(i_{20},j_{20}) \\ \equiv W_{AcyOp\cdot 0,1}(i_{10},j_{10}) \wedge W_{AcyOp\cdot 1,1}(i_{11},j_{11}) \wedge W_{AcyOp\cdot 2,1}(i_{12},j_{12}) \wedge W_{AcyOp\cdot 0,2}(i_{20},j_{20}) \end{array} \tag{14}$$

provided that

$$\begin{array}{l} \quad o_{10} = i_{21}\ ,\ p_{10} = j_{21}\ ,\ o_{11} = i_{22}\ ,\ p_{11} = j_{22} \\ \vdash \\ \quad (O_{AcyOp,1}(\langle o_{10},o_{11},o_{12}\rangle,\langle p_{10},p_{11},p_{12}\rangle\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \vee \\ \quad C_{AcyOp,1}(\langle o_{10},o_{11},o_{12}\rangle,\langle p_{10},p_{11},p_{12}\rangle\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle)) \Rightarrow \\ \qquad W_{AcyOp\cdot 1,2}(i_{21},j_{21}) \wedge W_{AcyOp\cdot 2,2}(i_{22},j_{22}) \end{array} \tag{15}$$

In (14), we see $W_{AcyOp,1;_{\delta}^{\circ}2}$ decomposed in the first equivalence into two contributions. One comes from $W_{AcyOp,1}$, and one from $W_{AcyOp\cdot 0,2}$, the contribution to $W_{AcyOp,2}$ coming from input signals i_{20},j_{20}. In the second equivalence, $W_{AcyOp,1}$ is further decomposed into subrelations for the individual inputs for $AcyOp_1$ and $AcyOp_2$.

The proviso in (15) stipulates that no matter whether the outcome of the first layer of the skew-sequential composition is realised via O or C, every such outcome is contained in the within relation for the second layer, i.e. $W_{AcyOpS,2}$. At this point this is an assumption which we demand of the system models involved, in order that the analysis to follow is well defined, i.e. so that whatever behaviour we see regarding both correct and faulty behaviour in the first layer, the output does not satisfy the retrenchment PO for the second layer 'spuriously' i.e. via the 'don't care' branch of the interpretation of the implication in the PO. Later, we will see that the assumption offers us no threat, since the within relations that we will use are all given by true. We write the assumption in *hypotheses* $\vdash$ *conclusion* style, in order to better show the intermediate values.

Note that, as in (14), in general there can be input values of $AcyOp_2$ that are not connected to output values of $AcyOp_1$ via ${}^{\circ}_{\circ\delta}$. In such cases, the occurrence of $W_{AcyOp,2}$ in (15) must be decomposed into the within relations for

individual abstract and concrete input pairs as in (13), and only those input pairs connected via $\overset{\circ}{\mathfrak{g}}_{\delta}$ would occur in (15). Any remaining input pairs would have to be unconstrained, or alternatively, one would have to rely on the environment to assert the necessary constraints, which is what (14) does.

Next we give the composed output relation:

$$\begin{aligned} &O_{AcyOp,1;_\delta 2}(\langle o_{12},o_{20},o_{21},o_{22}\rangle,\langle p_{12},p_{20},p_{21},p_{22}\rangle,\langle i_{10},i_{11},i_{12},i_{20}\rangle,\langle j_{10},j_{11},j_{12},j_{20}\rangle) \equiv \\ &\quad (\exists x_a,y_a,x_c,y_c \bullet O_{AcyOp,1}(\langle x_a,y_a,o_{12}\rangle,\langle x_c,y_c,p_{12}\rangle,\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \wedge \\ &\qquad\qquad O_{AcyOp,2}(\langle o_{20},o_{21},o_{22}\rangle,\langle p_{20},p_{21},p_{22}\rangle,\langle i_{20},x_a,y_a\rangle,\langle j_{20},x_c,y_c\rangle)) \end{aligned} \tag{16}$$

We see that this follows the pattern for skew-sequential composition of operations, so we can write it more succinctly as:

$$O_{AcyOp,1;_\delta 2} \equiv (O_{AcyOp,1} \overset{\circ}{\mathfrak{g}}_{\delta} O_{AcyOpS,2}) \tag{17}$$

Finally we give the concedes relation:

$$\begin{aligned} &C_{AcyOp,1;_\delta 2}(\langle o_{12},o_{20},o_{21},o_{22}\rangle,\langle p_{12},p_{20},p_{21},p_{22}\rangle,\langle i_{10},i_{11},i_{12},i_{20}\rangle,\langle j_{10},j_{11},j_{12},j_{20}\rangle) \equiv \\ &\quad ((\exists x_a,y_a,x_c,y_c \bullet O_{AcyOp,1}(\langle x_a,y_a,o_{12}\rangle,\langle x_c,y_c,p_{12}\rangle,\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \wedge \\ &\qquad\qquad C_{AcyOp,2}(\langle o_{20},o_{21},o_{22}\rangle,\langle p_{20},p_{21},p_{22}\rangle,\langle i_{20},x_a,y_a\rangle,\langle j_{20},x_c,y_c\rangle)) \vee \\ &\qquad (\exists x_a,y_a,x_c,y_c \bullet C_{AcyOp,1}(\langle x_a,y_a,o_{12}\rangle,\langle x_c,y_c,p_{12}\rangle,\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \wedge \\ &\qquad\qquad O_{AcyOp,2}(\langle o_{20},o_{21},o_{22}\rangle,\langle p_{20},p_{21},p_{22}\rangle,\langle i_{20},x_a,y_a\rangle,\langle j_{20},x_c,y_c\rangle)) \vee \\ &\qquad (\exists x_a,y_a,x_c,y_c \bullet C_{AcyOp,1}(\langle x_a,y_a,o_{12}\rangle,\langle x_c,y_c,p_{12}\rangle,\langle i_{10},i_{11},i_{12}\rangle,\langle j_{10},j_{11},j_{12}\rangle) \wedge \\ &\qquad\qquad C_{AcyOp,2}(\langle o_{20},o_{21},o_{22}\rangle,\langle p_{20},p_{21},p_{22}\rangle,\langle i_{20},x_a,y_a\rangle,\langle j_{20},x_c,y_c\rangle))) \end{aligned} \tag{18}$$

Again this can conveniently be abbreviated:

$$C_{AcyOp,1;_\delta 2} \equiv ((O_{AcyOp,1} \overset{\circ}{\mathfrak{g}}_{\delta} C_{AcyOp,2}) \vee (C_{AcyOp,1} \overset{\circ}{\mathfrak{g}}_{\delta} O_{AcyOp,2}) \vee (C_{AcyOp,1} \overset{\circ}{\mathfrak{g}}_{\delta} C_{AcyOp,2})) \tag{19}$$

We comment on the structure of (16) and (18) which both conform to a generic schema for combining retrenchments. Each pair of transitions that makes a retrenchment true can establish either the output or the concedes relation (provided the within relation holds), giving a disjunction of two possibilities. When n retrenchments are combined (which eventually produces a conjunction of the facts asserted by each), a disjunction of 2^n terms results via the distributive law. Of these 2^n terms, one will contain output relations exclusively and is deemed to be the output relation of the combination; the remaining $2^n - 1$ terms are deemed to form the concedes relation of the combination.

2.3.3. Associativity:

Although we do not prove it here, relying on [BJP08], we note that parallel and skew-sequential compositions of retrenchments are semantically associative, both individually, and when working together. The theory above is based entirely on sequential and parallel compositions of relations, so the associativity of relational compositions extends to the associativity of our manipulations of operations and retrenchments. The only nontrivial aspect of the latter is the associativity of the composition laws for retrenchment data, with the concomitant need to partition 2^n terms (in the general case of n retrenchments) into composed output and composed concedes relations. This aspect needs to be checked by explicit calculation, which confirms the validity of the procedure described for all n.

Looking ahead a little, the nature of the composition laws above means that while the output relation for a general circuit is most naturally viewed as monolithic (describing correct behaviour), the concedes relation is best seen as decomposing into a disjunctive normal form, each disjunct of which describes a separate kind of faulty situation.

2.4. Hierarchical Structure of Systems and Retrenchments

An important aspect of the design of large systems, is the ability to view them hierarchically. This means having a high level view in which certain features are described simply, and a lower level view in which they appear in greater detail. Mathematically, this requires a *refinement* process offering the kind of strong guarantees needed to ensure appropriate conformity between the levels — this being the self-same property that makes refinement too inflexible to describe fault injection below. Fortunately, retrenchment has been designed to co-exist fruitfully with refinement, so we can deal not only with the additional complexities coming from the fault analysis via retrenchment, but also the technical

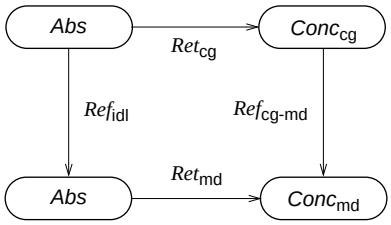

Fig. 2. The *Tower Pattern* instantiated for hierarchical design.

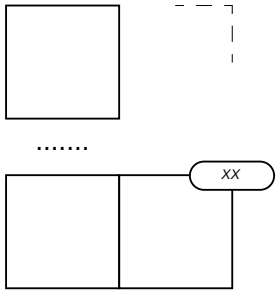

Fig. 3. Integrating new low level behaviour into the tower via system *XX*.

details of hierarchy, all in the same framework. Although there is a plethora of subtly different refinement techniques in the literature (see [dRE98] for a review of some of them), for us, it will be sufficient to characterise refinement via a degenerate form of the PO (4), in which:

- the concedes relation is (given by the predicate) false,
- the output relation relates output variables only (i.e. it does not contain any involvement of inputs).

The interworking of refinement and retrenchment is captured in the *Tower Pattern*, instantiated for our situation in Fig. 2. The *Tower Pattern* is essentially a commuting square of horizontal retrenchment rungs and vertical refinement columns, or it is an assembly of such squares stacked or abutted in the obvious way.

In Fig. 2, the two horizontal arrows are retrenchments: Ret_{cg} (a coarse grained retrenchment), and Ret_{md} (a more detailed retrenchment); these represent fault injection at two different levels of abstraction, a high level, and a lower, more detailed level respectively. The refinement Ref_{idl} on the left hand side represents an elaboration of the high level view of the ideal model into a low level ideal view, while the refinement $Ref_{\mathsf{cg\text{-}md}}$ on the right hand side, represents a similar elaboration of the faulty model. The fact that the square commutes represents the compatibility between ideal/faulty and high/low level views.

The tower, introduced in [BPJS05, BPJS06a, BPJS06b], and resting on fundamental existence theorems established in [BJ, Jes05], enables refinement developments of the same or related requirements, which are mutually incompatible from a refinement point of view, to be brought together into the same formal structure. The theorems of [BJ, Jes05] enable this to be done whenever we have any two adjacent edges of the square present; the theorems then 'complete the square', building the missing system and its relationships with the rest of the square, in an (at least semi-) automatic manner, opening the way to mechanical support.

The use of the tower enables the treatment of degrees of detail to be kept consistent with the fault injection process. In general, as one examines increasingly low level detail, it is possible that behaviours emerge that were not considered earlier. This can arise for a number of reasons. Not only does the original fault injection process itself require the use of the tower, but as the analysis of the various possibilities for failure proceeds, more detailed lower level possibilities can often emerge, and these have to be incorporated into the analysis in a consistent way. Indeed a staged introduction of different kinds of failure mode, if done consistently, can significantly aid clarity. To support such approaches, one needs further retrenchments to capture the new possibilities. This is illustrated in the 'L' shape of Fig. 3 in which the downward development has been extended to the right at the bottom with a new system *XX*, accommodating the fresh

behaviour not contained in the higher levels. To integrate these into the rest of the development, one can use the theory of the tower to lift the fresh behaviours of XX also to the highest levels if required; this is suggested by the dashed corner in the top right of Fig. 3.

3. Fault Tree Analysis

Fault Tree Analysis [VGRH81, VSD+02] is a well-established technique in safety and reliability assessment, whose purpose is to determine the conditions under which hazards can occur. Fault trees can support the decision-making process. They can be used not only as a diagnostic tool, but also to assist engineers in the evaluation of design alternatives or in carrying out design upgrades, an activity that may have an impact not only on safety, but also on resource allocation and design costs.

Fault tree analysis can be described as a deductive, analytical technique, whereby an undesired state (the so called *top (level) event* (TLE)) is specified, and the system is analyzed for the possible chains of *basic events* (e.g. system faults) that may cause the top event to occur. A Fault Tree (FT) makes use of logical gates to depict the logical interrelationships linking such events. The fault tree model is not in itself a quantitative model, but rather a qualitative model that can be evaluated quantitatively (e.g. to determine the probability of a safety hazard).

An example fault tree (redrawn from [VSD+02]) is depicted in Fig. 4.(a). The main symbols used in a fault tree are: square boxes, used to represent the top event (the topmost box); the *intermediate events* (the remaining boxes); and circles, used to represent the basic events. Logical gates, such as AND and OR, are used to link the events inside the tree. A plethora of symbols, not considered in the present paper, can be used to provide additional semantics (see [VSD+02] for a complete list). Special gates can be introduced to model timed dependencies, as in the so-called Dynamic Fault Tree methodology [DBB92].

In logical terms, the fault tree depicted in Fig. 4.(a) can be represented by the following logical formula: $(A \vee (B \vee C)) \wedge (C \vee (A \wedge B))$ with the intended meaning that a propositional symbol is true whenever the corresponding event occurs. Hence the top event occurs if and only if the formula evaluates to true. In strictly logical terms, fault trees can be considered equivalent if the associated logical formulae are equivalent. For instance, it is straightforward to see that the above formula is logically equivalent to $C \vee (A \wedge B)$, which can be graphically represented by the fault tree in Fig. 4.(b). This shape is of particular interest in reliability analysis, in that it represents the occurrence of a top event in terms of the so-called *minimal cut sets* (MCS). A minimal cut set can be seen as the smallest combination of component failures which causes the top event to occur. Both the previous trees have the same minimal cut sets, that is, C (single point of failure) and A, B (combination of two basic faults). Logically, a minimised fault tree such as the one in Fig. 4.(b) is associated with a Boolean formula in disjunctive normal form (i.e. a disjunction of conjunctions of propositional symbols). Minimal cut sets are of particular interest in reliability analysis because they represent simpler explanations for the top event, and they are often used as a starting point for quantitative analysis.

The exact way fault trees are generated in practice may vary, as described in [VGRH81, VSD+02]. A notable example of application of Fault Tree Analysis is given in SAE ARP 4761 [Int96] (Aerospace Recommended Practice), which describes a prototypical process for avionics, addressing the certification of civil aircraft. In short, the process follows the traditional 'V' shape, where the left-hand side of the 'V' represents the collection of the system requirements, and the right-hand side represents their validation. Fault trees can be produced at different stages of system validation. Important notions are the boundary of the analysis (e.g. whether analysis is performed at the subsystem or system level), and the level of resolution (e.g. abstraction and refinement techniques can be used, to assess a partially developed system, or to simplify the analyses). It is the responsibility of the safety engineer to decide, for a given top event, which are to be considered as the basic events, this depending on the boundary and level of resolution, and depending on which failures are considered relevant for the specific analysis.

Fault trees are developed starting from the top event. Causes which are considered to be elementary faults are developed as basic events, whereas the remaining causes are developed as intermediate events. This rule applies recursively to the intermediate events, which must in turn be traced back to their causes, until the tree is completely developed. In general, there is no unique way a fault tree can be built, in particular, there may be different choices for the intermediate events, and different ways to develop them. The guidelines given in [VSD+02] distinguish the case where a fault is localized to a given component ('state of component' fault) from the case where it is not ('state of system' fault). In the latter case, a fault is developed by considering its *immediate*, *necessary*, and *sufficient* causes for its occurrence. If the fault is localized to a given component, then its *primary*, *secondary* and *command* faults are investigated. Primary and secondary faults differ depending on whether the fault occurs in an environment for which the component is qualified (primary fault) or not qualified (secondary fault), whereas a command fault is due to a proper operation of a component, but at the wrong time or in the wrong place.

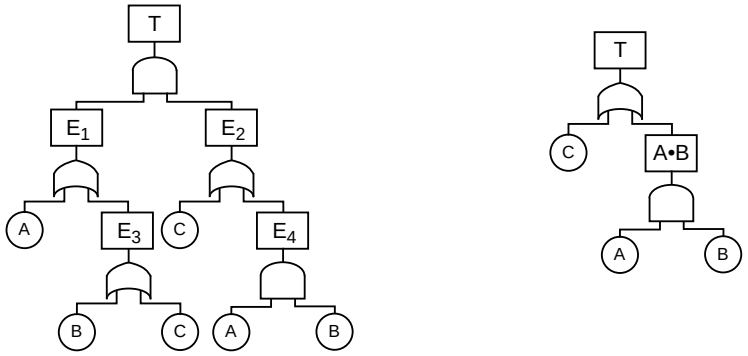

Fig. 4. An example Fault Tree (a), and a Minimised Fault Tree (b), corresponding to (a).

It is important to remark here that what makes the tree useful is not just the *logical relationship* between the events, but rather the way these events are *connected* (which is related to the notion of *causality*), together with a proper choice of intermediate events that are semantically relevant for safety engineers. In this sense, minimal cut set analysis does not exhaust the wide spectrum of possible questions which fault trees can answer.

The techniques described in this paper aim at improving on the ones presented in [BV07] (see Section 8 for a discussion of related work), which were tailored to the generation of flat fault trees, that is, to a graphical representation of minimal cut sets (in the style of Fig. 4.(b)). In this paper we present novel techniques for the mechanical generation of multi-level fault trees in the same style. The structure of these fault trees is built upon the underlying system structure, as given in the formal model. These techniques pave the way for automated generation of fault trees, as opposed to automated minimal cut set computation.

Concerning the semantics of the generated fault trees, one remark is in order. Delegating the discovery of *causes* of system events to an automated routine may appear to be (and to a large extent is) an ambitious goal (unless the routine is properly instructed by a human operator). What an automated routine can be reasonably asked to do is to find relationships of the form, *it is always the case that if event A occurs, then component B must have previously failed*. However, this does not authorize us to infer that there is a causal relation between events *A* and *B* (in the sense that the apparent relationships between events *A* and *B* might arise for other reasons, or just be a coincidence). Notwithstanding this difference, we show in the rest of this paper that automatically generated fault trees can still be informative enough to be useful for safety engineers, not least because we focus on *system structure*, so that the failure of a subsystem is related to the failure of components, both logically and causally.

4. Fault Injection and Fault Tree Structure by Example

In this section we examine fault injection via the retrenchment simulation relation in more detail, and illustrate, on an example, how the simulation relation may be manipulated to derive a fault tree in a mechanical manner. The discussion via example is made more precise in the theory of Section 5.

4.1. Fault Injection and the Retrenchment Simulation Relation

Consider an individual component of a system. We start with an ideal component, and wish to take into account possible faulty behaviour. The ideal behaviour is captured in the abstract transition relation Op_A, while the possible faulty behaviour is captured in the concrete transition relation Op_C. Normally, potentially faulty behaviour is an extension of correct behaviour, so, aside from changes of variable names in line with the conventions of Section 2, Op_A will be a subrelation of Op_C; we take this to be the case for the remainder of the paper, this being the sense in which our processes capture fault *injection*. We use this fact to simplify the retrenchment simulation relation (5) in the following way.

Firstly, we may assume that for each corresponding abstract/concrete pair of variables, both variables take their values in the same data type.[4] Next, we recall that the choice of retrenchment data for a retrenchment is a *modelling decision*. For our purposes, it will be sufficient to focus on retrenchments with the following properties:

1. The output relation is the statement that both abstract and concrete systems are displaying correct behaviour (expressed by saying that what has happened at the concrete level conforms to $Op_A(i,o)$ with concrete variables substituted for abstract ones):

$$O_{Op}(o,p,i,j) \equiv Op_A(i,o) \wedge j = i \wedge p = o \tag{20}$$

2. The concedes relation is the statement that while the abstract system is displaying correct behaviour (as it must), the concrete system *is failing* to do so. This is expressed by using an error relation $Err_{C,Op}$, a separately defined part of the definition of the concrete system, to capture those concrete transitions that are not merely the translations into concrete variables of abstract ones.[5]

$$C_{Op}(o,p,i,j) \equiv Op_A(i,o) \wedge Err_{C,Op}(j,p) \tag{21}$$

   The error relation must satisfy two conditions. Firstly, the error transitions are a subset of the concrete ones:

$$Err_{C,Op}(j,p) \Rightarrow Op_C(j,p) \tag{22}$$

   Secondly, all concrete transitions which are not ideal transitions (expressed using concrete variables) are error transitions:

$$Op_C(j,p) \wedge \neg Op_A(j,p) \Rightarrow Err_{C,Op}(j,p) \tag{23}$$

3. As mentioned earlier, the within relation is trivial (i.e. given by true). The reason for this is that as the faulty system's behaviour drifts increasingly away from the ideal, the discrepancy between the two can become arbitrarily large, so no *a priori* bound on the relationship between the inputs to the next component in the circuit can be imposed. This is consistent with all component behaviours being given by total relations, and with the remarks on within relations in Section 2.2.

This retrenchment design makes the simulation relation (5) decompose into independent abstract and concrete parts.

We can now discard the abstract part since it is subsumed (using concrete variables) in the concrete part.[6] Accordingly, we adjust the notation to reflect this. Both the retrenchment data and simulation relation become relations in concrete variables only. Thus the retrenchment data become:

$$W_{Op}(j) \equiv \mathsf{true} \tag{24}$$

$$O_{Op}(p,j) \equiv Op_A(j,p) \tag{25}$$

$$C_{Op}(p,j) \equiv Err_{C,Op}(j,p) \tag{26}$$

and the simulation relation absorbs the explicit statement of the transition and within relations, and merely decomposes the $(O \vee C)$ part into its $Op_A(j,p)$ and faulty pieces:

$$\Sigma^1 \equiv O_{Op}(p,j) \vee C_{Op}(p,j) \tag{27}$$

This is considerably simpler than the more general formulation, and aids efficiency in implementation.

4.2. An Example

We will base the rest of this section on an example. At the top of Fig. 5 we see a black-box depiction of a component *Fred*. *Fred* has two input signals and two output signals. The ideal version is $Fred_A$ and the potentially faulty version

[4] We do not state this formally, but it legitimises e.g. substitution of abstract by concrete variables, and equalities between abstract and concrete variables. Technically, this is just a convenience, to make some of the calculations in the sequel more transparent than they would otherwise need to be. As regards modelling though, it may be seen as a distortion, since it forces (the in principle) simpler abstract types to contain any exceptional values etc. that might be needed in the faulty concrete types. The distortion may be avoided at the expense of additional complexity in the formalisation, which would have to incorporate explicit notations for relations that mediated between the abstract and concrete types. Having these would, if anything, obscure the clarity of the technical exposition with burdensome detail, without adding anything of significance, so we avoided it.

[5] This strategy allows the same transition to be viewed as both correct and faulty, if necessary.

[6] We show below how a different retrenchment design would have prevented this simplification, without any benefit for fault tree generation.

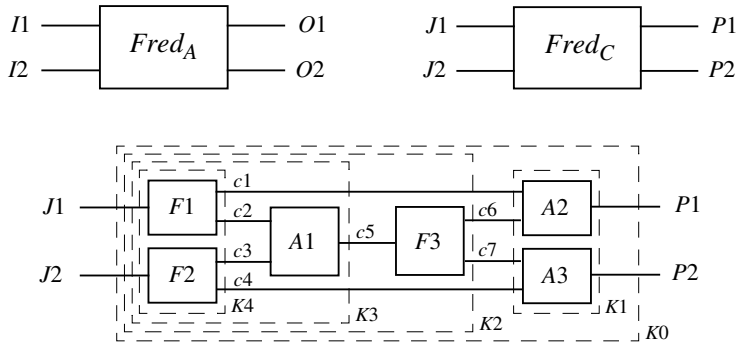

Fig. 5. A subsystem *Fred* and its internal structure.

is $Fred_C$, with I/O labelled according to our conventions. At the black-box level of abstraction, the only difference between $Fred_A$ and $Fred_C$ is in their transition relations, $Fred_A$ being a subset of $Fred_C$. Those transitions in $Fred_C$ which are not in $Fred_A$, i.e. those in $Err_{C,Fred}(j,p)$, represent faults of *Fred*, and can act as basic faults in a fault tree (FT) for some system level top level event (TLE) when the system description is such that *Fred* is regarded as a bottom level component.

Since *Fred* behaves instantaneously, we can write down the signature of *Fred*'s I/O transition relation, without recourse to any of *Fred*'s internal details:

$$Fred_C(\langle J1, J2\rangle, \langle P1, P2\rangle) \tag{28}$$

When *Fred* is *not* regarded as a bottom level component, but as something with internal structure, then the faulty transitions of *Fred* themselves become TLEs of a more detailed sub-FT which describes how they arise on the basis of the more detailed structure of *Fred*.

The bottom of Fig. 5 shows a more detailed view of $Fred_C$, in which it is a circuit where signals flow from left to right through components $A1, A2, A3, F1, F2, F3$, interconnected by wires $c1$-$c7$. We assume that all signals $J1, J2, P1, P2, c1$-$c7$ are of a fixed finite number of bits. Components $A1, A2, A3$ are adders. We assume that the adders do cutoff addition without overflow (so that any value greater than or equal to the maximum representable one is output as the maximum), e.g.:

$$A1_C(\langle c2, c3\rangle, c5) \equiv c5 = \min(c2 + c3, \mathsf{MAX}) \tag{29}$$

where MAX is the maximum representable value in the requisite number of bits. The number of bits is assumed sufficiently large that the cutoff effects do not occur in the examples we treat. For simplicity, we will assume in the rest of the paper that **adders never fail**. Therefore the transition relation for $A1_C$ is no different from that of $A1_A$, and similarly for $A2, A3$.

Elements $F1, F2, F3$ are two-output fanout nodes. Their ideal behaviour is to propagate their input value to their outputs, e.g.:

$$F1_A(I1, \langle a1, a2\rangle) \equiv a1 = a2 = I1 \tag{30}$$

Fanouts are assumed capable of failure. Their failure modes are *stuck_at_zero* faults on one or other of their outputs. Also for simplicity, **we assume that at most one of the *stuck_at_zero* faults is ever active for any fanout**. So the concrete transition relation for a fanout is given by e.g.:

$$F1_C(J1, \langle c1, c2\rangle) \equiv (F1.c1 \Rightarrow c1 = 0) \wedge (F1.c2 \Rightarrow c2 = 0) \wedge \neg(F1.c1 \wedge F1.c2) \;\; \textit{ELSE\_IDEAL} \tag{31}$$

In (31), $F1.c1$ ($F1$ output $c1$ *stuck_at_zero*) and $F1.c2$ ($F1$ output $c2$ *stuck_at_zero*) are (propositional) fault variables, which when true, 'switch on' the *stuck_at_zero* fault for the relevant output signal (this being another facet of fault *injection*).

In (31), *ELSE_IDEAL* represents the transliteration of the ideal $F1_A$ transition relation to J, P, c variables, and the mechanics of its being overridden by the faulty behaviour when either of $F1.c1$ or $F1.c2$ is true. When we unravel the details we get:

$$\begin{aligned} &F1_C(J1, \langle c1, c2\rangle) \equiv \\ &\quad (F1.c1 \wedge c1 = 0 \vee \neg F1.c1 \wedge c1 = J1) \wedge (F1.c2 \wedge c2 = 0 \vee \neg F1.c2 \wedge c2 = J1) \wedge \neg(F1.c1 \wedge F1.c2) \end{aligned} \tag{32}$$

Similar remarks apply to $F2, F3$.

In fault injection via fault variables, the truth of at least one fault variable is what singles out the subrelation $Err_{C,F1}$ from the rest of $F1_C$ in the formulation of Section 4.1:

$$Err_{C,F1}(J1, \langle c1, c2\rangle) \equiv F1_C(J1, \langle c1, c2\rangle) \wedge (F1.c1 \vee F1.c2) \tag{33}$$

Note that the presence of fault variables is not essential. One could simply remove them, making the concrete transition relation more nondeterministic, and defining $Err_{C,F1}$ by other means. Nevertheless, a description in terms of such variables is helpful for two purposes. Firstly, it allows an automated tool (e.g. the FSAP platform [BV07, FSA]) to keep track of the difference between ideal and faulty behaviour in a particularly simple way. Secondly, the propositional variables themselves can serve as names for the basic faults of the relevant components, labelling leaf nodes of FTs as required.

The ideal and faulty *Fred* models are related by a retrenchment. It will be sufficient to write down the retrenchment data for just the basic components, since the data for the overall system will emerge in a lazy fashion as needed via the retrenchment composition laws of Section 2.3 during the fault analysis below. We avail ourselves of the simplified forms for the retrenchment data derived above.

The adders are assumed fault-free. Thus for $A1$ we have:

$$W_{A1}(\langle c2, c3\rangle) \equiv \mathsf{true} \tag{34}$$

$$O_{A1}(c5, \langle c2, c3\rangle) \equiv c5 = c2 + c3 \tag{35}$$

$$C_{A1}(c5, \langle c2, c3\rangle) \equiv \mathsf{false} \tag{36}$$

$A2, A3$ are similar.[7] A particular consequence of this is that occurrences of $C_{A?}$ terms can be dropped below. For the fanout $F1$, we have:

$$W_{F1}(J1) \equiv \mathsf{true} \tag{37}$$

$$O_{F1}(\langle c1, c2\rangle, J1) \equiv c1 = c2 = J1 \tag{38}$$

$$C_{F1}(\langle c1, c2\rangle, J1) \equiv (F1.c1 \wedge c1 = 0 \wedge c2 = J1) \oplus (F1.c2 \wedge c2 = 0 \wedge c1 = J1) \tag{39}$$

where $\oplus$ is 'exclusive or'. In C_{F1}, we call the two disjuncts $C_{F1,c1}$ and $C_{F1,c2}$ respectively, i.e.:

$$C_{F1} = C_{F1,c1} \oplus C_{F1,c2} \tag{40}$$

Similar remarks hold for $F2, F3$.

4.3. Structured Fault Analysis

Finite acyclic circuits, such as the combinational logic circuits we are treating, possess a parsing which builds them up via parallel and skew-sequential composition. In general there will be several such parsings, which can be derived mechanically from a definition of the circuit in terms of elements and connections, or supplied manually. We work with one in which the elements closest to the inputs are the most deeply nested. Such a structure is convenient for a top-down fault analysis starting at the outputs, illustrated in the next section. For $Fred_C$, the structuring we use is illustrated by $K0$-$K4$ in Fig. 5.

Tied to the structure of the parsing, is the principal data that supports the analysis at the algorithmic level. For each basic component $A1, A2, A3, F1, F2, F3$ we have the output and concedes relations, held either explicitly or in a symbolic form, from which explicit tuples of the relations may easily be extracted. For the concessions there will be the decomposition into basic fault cases as in (40). For each compound entity $K0$-$K4$, the output and concedes relations are conjunctions or disjunctions of more basic component forms, so only the top level formulae are stored, with appropriate references to lower levels.

Fault analysis for a subsystem like *Fred* proceeds by taking a TLE, and deriving its causes by resolution with the retrenchment simulation relation (27). These causes are organised into a tree under the guidance of the structured

[7] This a good place to illustrate the consequences of different decisions about retrenchment data. A perfectly good alternative output relation for the adder $A1$ could be $O^{alt}_{A1}(a5, c5, \langle a2, a3\rangle, \langle c2, c3\rangle) \equiv a2 = c2 \wedge a3 = c3 \Leftrightarrow a5 = c5$. However deriving the kind of facts we require below via O^{alt}_{A1} would necessitate instantiating both abstract and concrete variables appropriately, and using the abstract and concrete transition relations to connect before- and after- values. This represents a considerable detour to arrive at an equivalent result. In another context, O^{alt}_{A1} might easily be preferable to O_{A1}, but not for the objective here.

retrenchment data for *Fred*, from which a FT can be extracted. A TLE for *Fred* is just a constraint on the values that some interface variables of $Fred_C$ can take (for $Fred_C$, the interface variables are $J1, J2, P1, P2$).

The goal is to unify the TLE with the simulation relation Σ^1. Since $\Sigma^1_{Fred} \equiv O_{Fred} \vee C_{Fred}$, this breaks into two subproblems. Normally the fault-free behaviour of a system is regarded as better understood than the faulty behaviour, so as an optimisation for expository convenience, we will assume that O terms such as $Fred_A = O_{Fred} = O_{K0}$ and O_{K1}-O_{K4} are all precomputed and available directly whenever required. They can always be calculated by the same backwards reasoning used for the concessions if necessary.

Let us fix a specific TLE: $J1 = J2 = P1 = 1$ (with $P2$ regarded as irrelevant) to illustrate the unification process. It is easy to check that this does not satisfy O_{Fred}. The analysis then proceeds 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 have to 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. (Note that the assumption of input readiness for all operations guarantees that a value can always be found for an unconstrained *output* without going to the trouble of actually calculating one; the assumption of finite data types guarantees that, if all else fails, all possible satisfying assignments may be found by brute force search.) In the following, the various steps are listed in a depth-first manner, for easier readability, but there is no requirement that the analysis is performed in this way; the only dependencies between the steps are data dependencies.

N.B. There can be significant tradeoffs between space and time complexity for a genuine algorithm, depending on the balance that is struck between depth-first and breadth-first aspects. In practice, any real algorithm must contain significant depth-first elements, since it is only when bottom level components are reached, that actual values can be assigned to variables. For expository economy, many of these detailed steps are finessed below. For the sake of theoretical simplicity, the finessing is carried to an extreme in Section 5 which makes extensive use of angelic nondeterminism.

TLE: To start with, $K0 = K2 \fatsemi K1$, so that $C_{K0} = O_{K2} \fatsemi C_{K1} \vee C_{K2} \fatsemi O_{K1} \vee C_{K2} \fatsemi C_{K1}$. Since we are working backwards through *Fred*, and $K1$ is nearest the outputs and is a compound structure, we first decompose $K1$, i.e. we decompose O_{K1} and C_{K1} into their component entities. Since $K1 = A2|A3$ and adders don't fail, C_{K1} is given by false, reducing C_{K0} to $C_{K2} \fatsemi O_{K1}$. Also $O_{K1} = O_{A2}|O_{A3}$. Now O_{A3} merely imposes existential constraints on $P2, c7, c4$ such that $A3_C(\langle c7, c4\rangle, P2)$ holds; we can put these to one side since the TLE does not constrain them further (and since input readiness of $A3_C$ ensures that $A3_C(\langle c7, c4\rangle, P2)$ can be satisfied for any $c7, c4$). Meanwhile, O_{A2} demands that $c1 + c6 = 1$ holds (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 $C_{F3} = C_{F3,c6} \oplus C_{F3,c7}$, and $C_{F3,c6}$ ($c6$ *stuck_at_zero*) is inconsistent with $c6 = 1$. Also O_{K3} (output of adder $A1$ with inputs $J1 = J2 = 1$ and correctly working fanouts $F1, F2$) forces $c5 = 2$, inconsistent with $c6 = 1$ too, so the terms containing these are dropped. So $C_{K3;F3}$ reduces to $C_{K3} \fatsemi O_{F3} \vee C_{K3} \fatsemi C_{F3,c7}$. In fact the distinction between these concerns only $c7$, whose precise value is immaterial, so only C_{K3} is of further interest. From $c6 = 1$, since fault variable $F3.c6$ is false, we deduce $c5 = 1$. So we can now decompose $C_{K3} = C_{K4;A1}$, which reduces to just $C_{K4} \fatsemi O_{A1}$ since adders don't fail. Now having $c5 = 1$ as adder output, implies $c2 = 0 \wedge c3 = 1$ or $c2 = 1 \wedge c3 = 0$, giving a disjunction into **TLE.L.L** or **TLE.L.R** for $C_{K4} \fatsemi O_{A1}$.

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 an exclusive or of two faults. However, we earlier derived $c1 = 0$, which is inconsistent with $J1 = 1$ and O_{F1}, eliminating a term and forcing $F1.c1$ true. But $c2 = 0$ (assumed to hold for this branch) forces $F1.c2$ true, and we have the constraint $F1.c1 \oplus F1.c2$ in C_{F1}, i.e. 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 cut set (i.e. valid cause) for the TLE.

TLE.R: We decompose C_{K2} as in case **TLE.L**, getting $O_{K3} \fatsemi C_{F3} \vee C_{K3} \fatsemi O_{F3} \vee C_{K3} \fatsemi C_{F3}$. 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 $O_{K3} \fatsemi C_{F3,c6}$ holds, with $c5 = 2$; or by case **TLE.R.2**, in which $C_{K3} \fatsemi O_{F3}$ is presumed to hold, with $c5 = 0$; or by case **TLE.R.3**, in which $C_{K3} \fatsemi C_{F3,c6}$ holds, with $c5$ as yet unconstrained; or by case **TLE.R.4**, in which $C_{K3} \fatsemi C_{F3,c7}$ is presumed to hold, with $c5 = 0$.

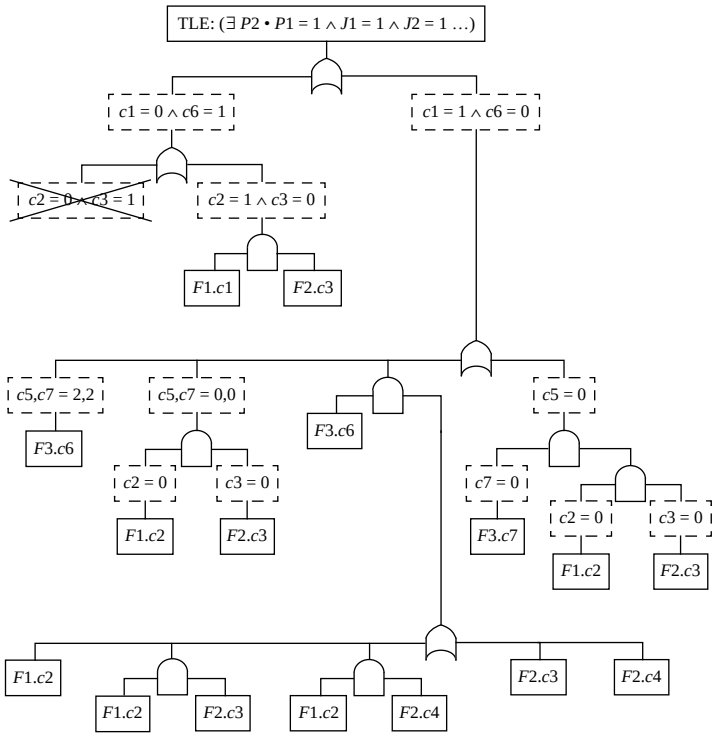

Fig. 6. Part of a Resolution Tree for the TLE of *Fred*.

TLE.R.1: $O_{K3}{}^{\circ}_{9}C_{F3,c6}$ holds, with $c5 = 2$. This is a valid cause of the TLE.

TLE.R.2: We have $C_{K3}{}^{\circ}_{9}O_{F3}$ and $c5 = 0$, so we decompose $C_{K3} = C_{K4;A1}$ which reduces to $C_{K4}{}^{\circ}_{9}O_{A1}$ since adders don't fail. From the adder, $c5 = 0$ implies $c2 = c3 = 0$ uniquely. The latter two imply $F1.c2$ and $F2.c3$ both true, which with $c1 = 1$ does not lead to a multiple failure for $F1$. Also $c4 = 1$ is acceptable for $F2$, leading to a cut set for the TLE.

TLE.R.3: We have $C_{K3}{}^{\circ}_{9}C_{F3,c6}$ 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 an exclusive or 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 $C_{K3}{}^{\circ}_{9}C_{F3,c7}$ and $c5 = 0$. The latter generates only one solution, i.e. $F1.c2$ and $F2.c3$ must both hold.

A tree that summarises the above is shown in Fig. 6. Near the top we show the variable assignments, but suppress them near the bottom to save space.

The resolution tree of Fig. 6 is the core output of our technique. As such it can serve as a starting point for subsequent processing of various kinds. In this paper, we are predominantly interested in fault trees —not only are fault trees a very familiar concept in safety analysis *per se*, but they also define the input format for commercial RAMS tools, see e.g. [ISO]— so we illustrate the post-processing needed to transform our resolution tree into a fault tree, focusing on a portion of Fig. 6.[8]

The main issues to attend to are: (i) to make sure that the basic faults occur only at the leaves of the FT, in round nodes, and (ii) to ensure that there are suitable intermediate events between any two logical connectives. We ensure (i) as a consequence of observing that basic faults occur only in concessions belonging to basic components. Since basic components occur in more complex subsystems only via parallel or sequential composition, and both kinds of

[8] 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).

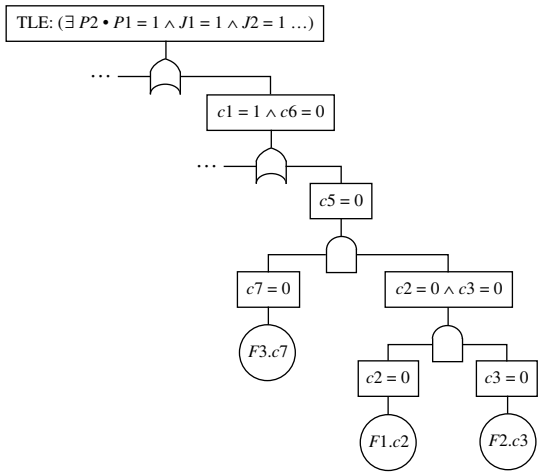

Fig. 7. Part of a FT for the TLE of *Fred*.

composition give concessions which are logically of the form $(O_1 \wedge C_2) \vee (C_1 \wedge O_2) \vee (C_1 \wedge C_2)$, it follows that whenever a C refers to a basic fault, we can always find a $\wedge$ immediately above it to hang the FT basic fault node from. We ensure (ii) by creating intermediate events whenever the analysis illustrated above generates adjacent connectives in the resolution tree construction. This is done in a bottom-up postprocessing phase, in which such intermediate events are created and labelled with the appropriate logical combinations of the immediate descendant variable assignment expressions (cf. the A●B node in Fig. 4.(b)). Such assignment expressions are bound to be available due to the bottom-up strategy, in contrast to the top-down strategy of the original construction which assigns to variables in the order that the resolution process dictates. We show the effect of these transformations for the rightmost branch of the resolution tree in Fig. 6, in the FT portion in Fig. 7.

5. Formal Fault Tree Derivation

In this section, we make precise an abstraction of the algorithm presented informally in the previous one, and prove various relevant properties. To ease comprehension, we relate the formal structures, where appropriate, to the running example *Fred* in the previous section.

5.1. Basic Definitions

To bring out the key points in the clearest manner possible, we reduce the technical complexity of our account by working under a number of simplifying assumptions regarding our systems and subsystems, as follows.[9]

Definition 5.1 (Basic Assumptions).

1. All systems and components are finite, as usual.
2. All variables are either *Fault Variables* (FVs), which are Boolean, or *I/O Variables* (IOVs), which take values in a single finite data type D.
3. A system S consists of a number of *Basic Components* (BCs), each of which has (at most) a single FV and a number of IOVs. When we have a particular system or subsystem T in mind, each IOV of T is either:
   (a) exclusive to one BC, say A, of T, and is used by A (as either an input variable or an output variable, but not both) for communicating with the external environment of T, and is called an *External Variable* (XV) of T; or

[9] In particular, we restrict to at most one possible fault per basic component, immediately ruling out our running example from Section 4. However, this just reduces theoretical clutter, without undermining any point of importance.

(b) is shared by two BCs, say A_1 and A_2, of T, and is used for communicating between them, being an input for one (A_1 say) and an output for the other (A_2), and is called an *Internal Variable* (IV) of T and an *Interface Variable* (IFV) of A_1 and A_2.

The IVs and XVs of S are called the top level IVs and XVs (TLIVs and TLXVs).

4. The I/O transformer of a BC A is given by:

$$A(j,p) = (\neg A.F \wedge O_A) \vee (A.F \wedge C_A) \tag{41}$$

where $A.F$ is the FV of A; j,p are the inputs and outputs, and O_A, C_A are the output and concedes relations of A respectively, defined as in Section 4.1.

5. A TLE for a system S is a quantifier-free expression in TLXVs. N.B. The finite data type assumption enables us to eliminate quantifiers from TLEs via enumeration: e.g. if $D = \{0,1\}$, the expression $(\exists P_2 \bullet X(J_1, \langle P_1, P_2 \rangle) \wedge J_1 = P_1)$ can be rewritten as $X(J_1, \langle P_1, P_2 \rangle) \wedge ((J_1 = 0 \wedge P_1 = 0 \wedge P_2 = 0) \vee (J_1 = 0 \wedge P_1 = 0 \wedge P_2 = 1) \vee (J_1 = 1 \wedge P_1 = 1 \wedge P_2 = 0) \vee (J_1 = 1 \wedge P_1 = 1 \wedge P_2 = 1))$.

In the context of the *Fred* example, $A1, A2, A3, F1, F2, F3$ are the BCs, with only $F1, F2, F3$ having FVs, as described earlier. Regarding the IOVs, $c5$ is an IV of subsystem $K2$ (which includes both of the BCs that share $c5$, i.e. $A1$ and $F3$), but is an XV of subsystem $K3$ (which includes only one of them, namely $A1$). The TLXVs are $J1, J2, P1, P2$, and all other IOVs are TLIVs.

Definition 5.2 (Assignment). An assignment is an equation of the form $(\mathit{variable} = \mathit{value})$, or a conjunction of such formulae.

Definition 5.3 (Valuation Set). Let $V = \{x_1, \ldots, x_k\}$, with $k \geq 0$, be a set of variables ranging over the finite domain D. We define the valuation set of D with respect to V, denoted $VS(D,V)$, as follows:

$$VS(D,V) = \bigcup_{g:V \to D} \{ \bigwedge_{i=1}^{k} (x_i = g(x_i)) \} \tag{42}$$

(For instance $VS(\{0,1\}, \{x_1, x_2\}) = \{x_1 = 0 \wedge x_2 = 0, x_1 = 0 \wedge x_2 = 1, x_1 = 1 \wedge x_2 = 0, x_1 = 1 \wedge x_2 = 1\}$.)

Definition 5.4 (Fault Variables). Given a system S, we denote by $FV(S)$ the set of fault variables of S.

Definition 5.5 (Fault Configuration). Let S be a system. A fault configuration for S is a subset $FC \subseteq FV(S)$. Furthermore, we denote by $\overline{FC}$ the set $(FV(S) - FC)$.

In particular, an arbitrary fault configuration need not correspond to any possible fault of the system; it is just a set of fault variables.

Definition 5.6 (Structural Expressions). Structural expressions K (for a system S) are defined by the following grammar:

$$\begin{array}{lcl} K & ::= & BC \mid (K \fatsemi K) \mid (K \mid K) \\ BC & \in & \{A \ldots\} \end{array} \tag{43}$$

where K is the non-terminal of the language (used to refer to substructures of S), and BC is a meta-notation referring to an element of the set $\{A \ldots\}$ of basic component names of S.

A sentence for S of the grammar is *ground* if it consists solely of basic components and combinators $\fatsemi$ and $|$, and, viewed bottom-up, each combinator $\fatsemi$ or $|$ corresponds to an 'instantaneous' parallel or skew-sequential composition of the appropriate subsystems of S. N.B. (43) ensures that each basic component A of S is enclosed in its own individual K substructure in the parse tree of a ground sentence.

Definition 5.7 (Parse Tree Node Attributes). To each K node of the parse tree of a ground sentence for a system S are associated a number of things:

1. An index (or other unique attribute) to uniquely identify it, e.g. $K3$.
2. An output and a concedes relation name for (i.e. a reference to) the appropriate substructure of S e.g. O_{K3}, C_{K3}.
3. An output and a concedes relation body for the appropriate substructure of S. For a $\fatsemi$ or $|$ K node n, these are given by formulae (in the output and concedes relation names for (i.e. references to) child nodes of n) for the parallel and

skew-sequential compositions of retrenchment data, given earlier. For a *BC K* node, these are the relations in (41).

To each *BC* node of the parse tree of a ground sentence we associate:

4. An index (or other unique attribute) to uniquely identify it, e.g. $A19$.
5. The name of the FV for the basic component, e.g. $A19.F$.

Thus each *K* node of the parse tree of (the structure of a system) *S* has a unique identifier, and, in effect, two equations (equating output and concedes relation names to their bodies), which are unwound in exploring the structure step by step. Each *BC* node has a unique identifier and the appropriate fault variable. We assume that whenever a skew-sequential composition occurs, the IFVs are understood from context.

It is clear that we have formalised the description illustrated in Fig. 5 (aside from showing each BC inside an individual *K*, which is suppressed). For example, $(K2 \fatsemi K1)$ is obtained when K_{Fred}, the root of the parse tree of *Fred*, is expanded one level according to the structure of Fig. 5. $K_{Fred}, K1$ and $K2$ are non-terminals in the parse tree, and the signals $c1, c6, c7, c4$, are the IFVs of the sequential composition of $K1$ and $K2$. $O_{Fred} = (O_{K2} \fatsemi O_{K1})$ and $C_{Fred} = O_{K2} \fatsemi C_{K1} \vee C_{K2} \fatsemi O_{K1} \vee C_{K2} \fatsemi C_{K1}$ are the two equations associated with K_{Fred}. $A1$ and $A2$ are BCs of *Fred*, with O_{A1} the output relation of $A1$, given by (35).

Definition 5.8 (Cut Set). Let *S* be (the I/O transformer of) a system with TLXVs J, P. Let $FC \subseteq FV(S)$ be a fault configuration, and *TLE* a top level event. We say that *FC* is a cut set for *S* and *TLE* if there exist values for *J* and *P* such that:

$$S(J,P) \wedge TLE \wedge \bigwedge_{F \in FC} F \wedge \bigwedge_{G \in \overline{FC}} \neg G \tag{44}$$

is true (where an empty conjunction is as usual true).

Thus, unlike a fault configuration, which is relatively arbitrary, a cut set (with respect to some understood TLE) must be a set of fault variables whose truth exactly captures some consistent system behaviour.

Definition 5.9 (Realisability). Let Φ be an expression in some variables of a system *S*. Then Φ is *realisable* in *S* iff values can be found for all as yet uninstantiated variables of *S* such that (44), with Φ replacing *TLE* and with *FC* defined implicitly by the truth/falsehood of fault variables, becomes true.

Since an arbitrary candidate TLE need not *a priori* conform to any consistent system behaviour, we conclude that a TLE is realisable in *S* iff there is a cut set for *S* and *TLE*.

5.2. The Resolution Tree Algorithm

We now introduce various kinds of tree which will carry the fruits of our analysis. The previously introduced parse tree of the system structure, provides the backbone upon which the construction of these new trees depends. The first tree whose construction we introduce is the resolution tree (RT). This is the generic version of what the tree in Fig. 6 for the running example roughly corresponded to. (We say 'roughly', since Fig. 6 showed one 'blind alley' (which forced the backtrack) and suppressed a good deal of the detail lower down.) Once we have discussed its construction, we prove a number of properties of interest for these trees. Subsequently, RTs are transformed into fault trees (FTs) in a number of stages.

We start by describing the kinds of node that these trees consist of. In each case we give the node tag (TLE, AND, OR, etc.), and, where needed, the additional information attached to the node following the colon. The last node type (the IE node) is not needed immediately, but is listed here for convenience.

Definition 5.10 (Node Types). All trees will consist of nodes tagged with one the following labels. Nodes (other than 'AND' and 'OR' nodes) contain additional label-specific information as described.

1. TLE(*TLE*): A **T**op **L**evel **E**vent node containing a top level event *TLE*.
2. AND: A conjunction node.
3. OR: A disjunction node.
4. ASG(Φ): An **AS**si**G**nment node containing an element $\Phi \in VS(D,V)$ of the valuation set of *D* with respect to a set of variables *V* (*V* will usually be the IFVs of a skew-sequential composition referred to by a *K* node of the parse tree of *S*).

Function *GenRT*
Input: TLE, S, a structure for S
Output: A resolution tree for *TLE* given by *root*
Begin
1 **If** $O_S \wedge TLE$ is not satisfiable and $C_S \wedge TLE$ is not satisfiable
2 **Then** Create a tree as in Fig. 9.(a). (The tree contains just *TLE*.)
3 **ElseIf** $O_S \wedge TLE$ is satisfiable and $C_S \wedge TLE$ is not satisfiable
4 **Then** Create a tree as in Fig. 9.(b).
(The GOAS child of *root* contains an assignment for all TLXVs and TLIVs of S that makes $O_S \wedge TLE$ realisable.)
5 **Else** ($C_S \wedge TLE$ is satisfiable)
6 **Let** $\{\theta_1 \ldots \theta_n\} = \Theta_S \subseteq VS(D, TLXVs(S))$ be the assignments to the TLXVs of S which make $C_S \wedge TLE$ realisable
7 **In If** $O_S \wedge TLE$ is not satisfiable
8 **Then** Create a tree as in Fig. 9.(c).
9 **Else** Create a tree as in Fig. 9.(d).
(The GOAS node contains an assignment for all TLXVs and TLIVs of S that makes $O_S \wedge TLE$ realisable.)
10 **EndIf**
11 **ForAll** $\theta_k \in \Theta_S$
12 **Do** $Expand(n_{k1})$. (The n_k : ASG node contains θ_k. The n_{k1} : C node contains concedes relation name C_S.)
13 **EndForAll**
14 **EndLet**
15 **EndIf**;
16 **return** *root*;
End

Fig. 8. The algorithm for generating the Resolution Tree.

5. OAS(Φ): An *O*-**AS**signment node containing an element $\Phi \in VS(D, IVs)$ of the valuation set of D with respect to the IVs of a subsystem of S referred to by a K node of the parse tree of S (used when the output relation O_K of K is satisfiable).
6. GOAS(Φ): A **G**lobal *O*-**AS**signment node containing an element Φ of the valuation set of D with respect to all TLXVs and TLIVs of S (used when O_S itself is satisfiable).
7. BF(*FV*): A **B**asic **F**ault node containing the fault variable *FV* of a BC of S.
8. $C(C_K)$: A **C**oncedes node containing a concedes relation name C_K for a substructure of S referred to by a K node of the parse tree of S.
9. IE(Ψ): An **I**ntermediate **E**vent node containing a Boolean combination Ψ of assignments of values to IFVs of S.

Of these node types, the first three should be self-explanatory. The next type, the ASG(Φ) node, typically contains an assignment (in the sense of Def. 5.2) of the interface variables of a skew-sequential composition of subsystems, to some values that make the rest of the analysis non-void (in other words, to some values that maintain the realisability of the analysis-so-far). Under an engineering perspective, comparing with traditional fault tree analysis, such nodes correspond to genuine intermediate events that relate the interface variables of different components. The next node type, the OAS(Φ) node, again contains an assignment, but this time of the internal variables of a subsystem, to some values that witness that the subsystem is operating in a fault-free manner (i.e. such that its output relation O becomes true, again maintaining the realisability of the analysis-so-far). Under an engineering perspective, such nodes could be modelled using *external events* (also called *house events*, compare [VSD$^+$02]) or they could simply be discarded, given that the relevant assignments specify fault-free behavior of a subsystem. They are important for the completeness of the resolution process, but when resolution trees get post-processed to fault trees of a standard format, they get eliminated (see Section 5.3). The next type, the GOAS(Φ) node, is like an OAS node, but applied to the whole system and not just some subsystem; it takes care of the completely fault-free case. Again, in a traditional fault tree, it would be modelled with an external event since it corresponds to an empty cut set. The next types, the $C(C_K)$ and BF(*FV*) types, deal with faulty cases. The former type, $C(C_K)$, contains (a reference to) a concedes relation. It is just a placeholder, being used during the working of the resolution tree algorithm. In fact, these nodes are not preserved in the final resolution tree, being introduced, but ultimately also eliminated, as the construction proceeds. The latter type, BF(*FV*), contains what is effectively a degenerate version of the former, namely a basic fault variable. In engineering terms, it corresponds to a genuine basic event, modeling a 'state of component' fault in the terminology of Section 3 (this being either a primary or a secondary fault — compare also the discussion in Section 9). The last node type, the IE(Ψ) type, contains a Boolean combination of assignments of the kind already discussed, and it can interpreted as modelling an intermediate event.

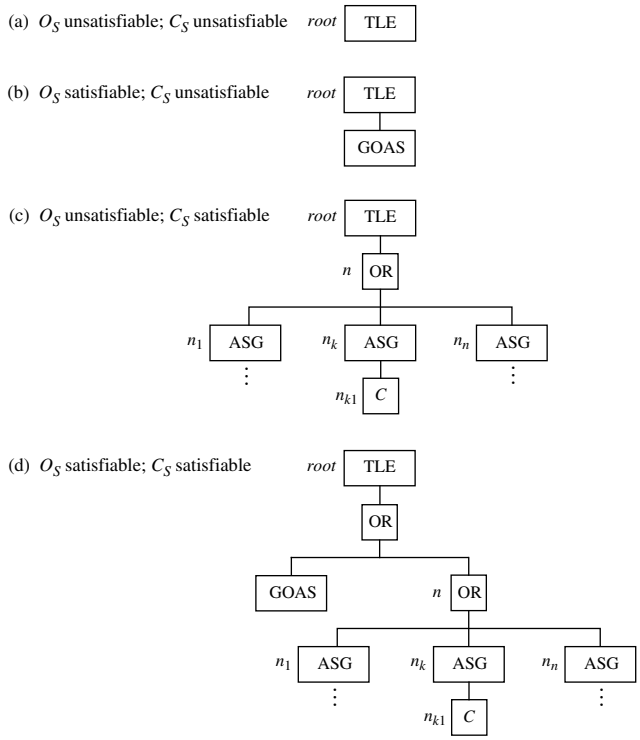

Fig. 9. Cases for the *GenRT* function.

In the algorithms which follow we make considerable use of *angelic nondeterminism* for discovering satisfying assignments (of the kind noted) for expressions at the point that they are needed in building the tree. This considerably simplifies the presentation, and shows that despite the appearance of the pseudocode, we are still at a relatively high level of abstraction. In practice of course, at the implementation level, some sort of depth first exploration with backtracking would be needed to achieve the required effect.

Definition 5.11 (Resolution Tree). Let S be a system and *TLE* a top level event for S. Assume a structuring for S given by a parse tree of a ground sentence of the grammar (43), together with its associated labels and equations. Then the algorithm of Fig. 8 generates a *resolution tree* for *TLE* with respect to S and the given structure.

As is clear by inspection, the resolution tree algorithm of Fig. 8 works top down, guided through the structure of the system by the structure of the parse tree which has been used to organise the system's structure. Every time some variable values are needed on the interface between two subsystems, so that the analysis can separately branch into the two subsystems, they are discovered angelically in the **Let** statement on line 6 of Fig. 8 (similarly on line 6 of Fig. 10), so as not to clutter the algorithm presentation with low level details. Note that for engineering convenience, the algorithm of Fig. 8 distinguishes at top level all four cases in which the TLE is satisfiable or not via O_S and/or C_S (i.e. all four combinations of possibly completely fault free, and possibly faulty behaviour), so that this structure can persist into the ultimate FT.

Referring back to our running example, we see that Fig. 6 corresponds to a RT whose top level structure is of type (c) in Fig. 9 — the TLE is neither unsatisfiable, nor is it satisfiable through fault-free behaviour, eliminating cases (a), (b) and (d).

The top level function *GenRT* makes use of a recursive function $Expand(n)$ to grow the tree from C nodes until the leaves of the tree are generated. This is given in Fig. 10. In Fig. 10, the phrase '*m-root-realisable*' is an abbreviation for 'realisable, and is consistent with all assignments in all nodes on the path from *m* back to *root*' (and including both). In engineering terms, we can draw a parallel between the expansion routine, and the 'state of system' and 'state of component' decomposition rules prescribed by [VSD+02] (compare Section 3 and also the account in Section 9).

Proposition 5.12 (Termination). The algorithm in Fig. 8 always terminates.

Proof. Given that the system S is acyclic, then the algorithm trivially terminates because each expansion step corresponds to unfolding the parsing of the system. □

The RT is important because it examines *all* the system variables, to ensure that only consistent system behaviours are generated. However, once the RT algorithm has produced its output, we can use the resulting RT to focus on just the causes of faulty behaviours of S. We call these the (computed) cut sets, and they are intended to correspond to the fault configurations/cut sets defined earlier. (Eventually, of course, we will show that they are the same.) Since we have a tree structure to work with already (the RT), the only sensible approach is to use an inductive strategy. The next few definitions handle the details.

Definition 5.13 (Nearest Relevant Descendant Nodes). Let RT be a resolution tree, and n be a node in RT. By a *Nearest Relevant Descendant Node* (NRDN) of n we mean the first proper descendant node tagged with one of GOAS, OR, AND, BF on some path from n towards a leaf of RT. The *Nearest Relevant Descendant Node Set* (NRDNS) of n is the set of all NRDNs of n.

Definition 5.14 (Computed Cut Node Sets). Let S be a system, and TLE a top level event. Let $RT = GenRT(TLE, S)$, and let n be a node of RT tagged with one of TLE, GOAS, OR, AND, BF. Let N be the subtree of RT rooted at n. Then the set of computed cut node sets of N denoted $CCNS(N)$ or $CCNS(n)$ as convenient, is defined as follows:

$$CCNS(N) = \begin{cases} \varnothing & RT \text{ has just a TLE node} \\ \{\{n\}\} & n \text{ is a GOAS node, or a BF node for BC } A \\ \bigcup_{n_i \in NRDNS(n)} CCNS(n_i) & n \text{ is an OR node, or a TLE node} \\ \{\bigcup_{i=1}^{k} CCNS_i \mid CCNS_i \in CCNS(n_i)\} & n \text{ is an AND node, and } NRDNS(n) = \{n_1 \ldots n_k\} \end{cases}$$

As we see from the above equation, at each point of the inductive computation, there are four possible cases. The first case deals with inconsistency — no cut node set can explain the TLE. The second case deals with leaves of the tree (tag BF or GOAS), whose associated set of cut node sets has just one element, the singleton cut node set $\{n\}$ — for a GOAS node, this indicates that nominal behaviour can explain the TLE, while for a BF node, it means that the constraints at that leaf of the RT can be satisfied by invoking that particular basic fault. The third case is for OR nodes. For these, any alternative cut node set from any of the nearest relevant descendant nodes (NRDNs) of the OR node will do as an explanation of the constraints demanded at that point; i.e. any alternative is a cut node set of the OR node itself. So the set of cut node sets for the OR node is just the union of those for its NRDNs. The fourth case covers AND nodes. For these, a cut node set consists of a contribution from each of its NRDN nodes, explaining the constraints demanded at the relevant subtree. So the set of cut node sets for the AND node consists of the set of all possible ways of combining a selection from each of the NRDNs. A TLE node will in general have a set of faulty causes that is elaborated via a descendant OR node.

So we have identified the nodes of the RT that can explain the TLE. But safety analysis is more interested in faults than nodes, so the next definition replaces each basic fault node by its corresponding fault variable (which amounts to the name of the fault in the present framework), yielding a cut set. A GOAS node is replaced by the empty set since it attests to explaining the TLE by fault-free behaviour.

Definition 5.15 (Computed Cut Sets). Let S be a system, and TLE a top level event. Let $RT = GenRT(TLE, S)$, and let n, the root of subtree N, be a node of RT tagged with one of TLE, GOAS, OR, AND, BF. Let $CCNS(N)$ be the computed cut node sets for N. Then the computed cut sets for N, denoted $CCS(N)$ (or $CCS(n)$ when convenient), are generated by replacing each BF node bf in $CCNS(N)$ by the FV, say $A.F$, that it contains, and the GOAS node, if present, by $\varnothing$.

Proposition 5.16 (CCNS for CCS). If S is a system, TLE a top level event, and $CS \in CCS(N)$ is a computed cut set for N, with N a subtree of $RT = GenRT(TLE, S)$ as in Def. 5.15, then there is a(t least one) computed cut node set in $CCNS(N)$ that generates CS. Any such computed cut node set is called a cut node set for CS.

We want to confirm that, in finding a cut set, we have found a consistent behaviour of the entire system. One fact that contributes to this is knowing that all system variables have been assigned to. This is not completely trivial to check, since in general, a cut node set includes only *some* leaves of the RT, due to the presence of OR nodes.

Function *Expand*(n)
Begin
1 **If** $n : C$ is a BF node associated with a $K \equiv BC$ node of S
2 **Then** Replace $n : C$ with a subtree as in Fig. 11.(a). (The n : ASG node contains an assignment to the outputs of the BC which makes C *n-root-realisable*. The n_1 : BF node contains the FV for the BC in question.)
3 **ElseIf** $n : C \equiv O_1|C_2 \vee C_1|O_2 \vee C_1|C_2$ associated with a $K \equiv K_1|K_2$ node of the structure of S
4 **Then** Replace $n : C$ with a subtree as in Fig. 11.(b). (The n_1 : OAS node contains an assignment to the IVs of component K_1 which makes $O_1 \wedge C_2$ *n-root-realisable*. The n_2 : OAS node contains an assignment to the IVs of component K_2 which makes $O_2 \wedge C_1$ *n-root-realisable*. If any one branch n_1, n_2, n_3 is not *n-root-realisable*, it is pruned. If two branches are not *n-root-realisable*, they are both pruned (though the OR node remains).) As applicable, *Expand*(n_{11}); *Expand*(n_{21}); *Expand*(n_{31}); *Expand*(n_{32}).
5 **ElseIf** $n : C \equiv O_1 \,{}_{\S}\, C_2 \vee C_1 \,{}_{\S}\, O_2 \vee C_1 \,{}_{\S}\, C_2$ associated with a $K \equiv K_1 \,{}_{\S}\, K_2$ node of the structure of S
6 **Then Let** $\{\theta_1 \ldots \theta_n\} = \Theta_K \subseteq VS(D, IFVs(K))$ be the assignments to the IFVs of the composition $K \equiv K_1 \,{}_{\S}\, K_2$ which make the concession C_K of K *n-root-realisable*
7 **In** Replace $n : C$ with a subtree as in Fig. 11.(c).
8 **ForAll** $\theta_k \in \Theta_K$
9 **Do** (The n_k : ASG node contains θ_k. The n_{k11} : OAS node contains an assignment to the IVs of K_1 which makes $O_1 \wedge C_2$ n_{k1}-*root-realisable*. The n_{k12} : OAS node contains an assignment to the IVs of K_2 which makes $O_2 \wedge C_1$ n_{k1}-*root-realisable*. If any one branch $n_{k11}, n_{k12}, n_{k13}$ is not n_{k1}-*root-realisable*, it is pruned. If two branches are not n_{k1}-*root-realisable*, they are both pruned (though the OR node n_{k1} remains).) As applicable, *Expand*(n_{k111}); *Expand*(n_{k121}); *Expand*(n_{k131}); *Expand*(n_{k132}).
10 **EndForAll**
11 **EndLet**
12 **EndIf**;
13 **return**;
End

Fig. 10. The *Expand* function for the Resolution Tree.

Definition 5.17 (Cut Path). Let S be a system, *TLE* a top level event, and let $CS \in CCS(RT)$ be a cut set computed from $RT = GenRT(TLE, S)$. Let $CNS \in CCNS(RT)$ be a cut node set for CS. If $n \in CNS$, then the cut path of n with respect to CNS is the set of nodes along the path from n back to the root of RT (and including both). The cut path set (CPS) of CS with respect to CNS is the union of the cut paths for all $n \in CNS$.

Proposition 5.18 (Assignment Totality). Let S be a system, let *TLE* be a top level event, let $CS \in CCS(RT)$ be a cut set computed from $RT = GenRT(TLE, S)$ and generated by a cut node set CNS. Then the CPS of CS with respect to CNS contains an assignment to every system non-fault variable.

Proof. Since $CS \in CCS(RT)$, we have $CCS(RT) \neq \varnothing$. So the RT contains more than just the TLE node. If $CS = \varnothing$, then this cut set can only come from a GOAS node, which assigns to all the system non-fault variables. Since the GOAS node is in the CPS of CS with respect to CNS, we are done. Otherwise CS is a nonempty set of FVs and we have to show that its CPS with respect to CNS contains assignments to all system non-fault variables.

It is clear in such a case that C_S is satisfiable, and that $GenRT(TLE, S)$ therefore creates a top level subtree as in cases (c) or (d) of Fig. 9. We focus on the subtree rooted at the OR node n, the parent of the ASG node(s) in these cases — the subtrees below n in (c) and (d) are identical. It is clear that $CS \neq \varnothing$ implies that $CS \in CCS(RT)$ implies that $CS \in CCS(n)$. Now Defs. 5.14 and 5.15 show that CS and its CPS come from a single branch of the disjunction under n. This branch descends through an ASG node (containing an assignment to all TLXVs), to a C node developed by *Expand*. Since $CS \neq \varnothing$, there must be at least one cut path in the CPS of CS with respect to CNS. Any such path must pass through the ASG node, and so the CPS of CS with respect to CNS is guaranteed to contain an assignment to all TLXVs. It is thus sufficient to show that the relevant top level call to *Expand* develops a subtree such that all cut paths in the CPS of CS with respect to CNS descend into this subtree, and all TLIVs are assigned to in the CPS.

Consider the parse tree of S, where the detailed structure under any non-terminal corresponding to a non-failing subsystem (i.e. a subsystem K_k for which O_{K_k} is true) has been truncated. We proceed by induction on this truncated tree, with the following induction hypothesis.

> If K is a node for which C_K is true, N is a subtree of $RT = GenRT(TLE, S)$ generated by *Expand* on C_K, CS_K is a cut set in $CCS(N)$, and CNS_K is a cut node set generating CS_K, then all paths of the CPS of CNS_K descend into N and all IVs of K are assigned to in the CPS.

To deduce the required conclusion, we apply the hypothesis to the root of the parse tree of S, truncated in a manner corresponding to the realisation of C_S that yielded CS.

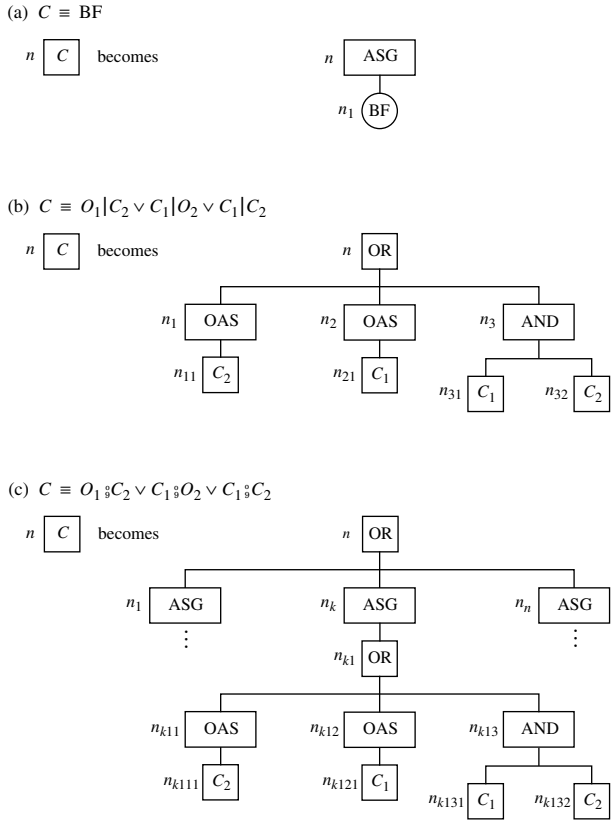

Fig. 11. Cases for the *Expand* function.

Base cases: $K \equiv$ BC, a single BC A for which C_K is true. By case (a) of Fig. 11, the call to *Expand* creates a subtree with a BF leaf containing the FV of the BC, and an ASG node, which assigns to the BC's outputs. The BF node contains the FV of A, i.e. $A.F$, so by Defs. 5.14 and 5.15 the generated cut set is $\{A.F\}$. Evidently the single cut path in the CPS of K descends into the subtree generated by *Expand*, and since a BC has no IVs, this CPS trivially contains an assignment to every IV of the BC. If K is a non-faulty subsystem, then there is nothing to prove.

Inductive steps: $K \equiv K1|K2$. Then the IVs of K partition into the IVs of $K1$ and of $K2$. If $K1$ is a non-failing subsystem and $K2$ is a failing subsystem, then the cut set of K is just the cut set of $K2$. By case (b) of Fig. 11, the relevant call to *Expand* creates a branch containing an OAS node for $K1$, descending to a subtree for $K2$. The OAS node assigns to all the IVs of $K1$. By the inductive hypothesis, all paths of the CPS of the cut set of $K2$ descend into the $K2$ subtree, and all IVs of $K2$ are assigned to in the CPS. Since the cut paths in the CPS of $K2$ must ascend through the OAS node, the CPS for the cut set of K contains assignments to all IVs of K, as required. Similarly if $K1$ is a failing subsystem and $K2$ is a non-failing subsystem.

If $K1$ and of $K2$ are both failing subsystems, the cut set of K partitions into nonempty cut sets for $K1$ and for $K2$, as does the set of IVs of K. The call to *Expand* creates an AND of the subtrees for $K1$ and for $K2$, and the CPS thereby also partitions into nonempty CPS for the $K1$ and $K2$ subtrees. We then use the inductive hypothesis twice, after which Defs. 5.14 and 5.15 stipulate that the AND node forces the union of the cut subsets, the cut node subsets, and their CPSs, ensuring that all IVs of K are assigned to as needed.

$K \equiv K1 \wr K2$. Then the IVs of K partition into the IFVs of the composition, plus the two sets of IVs of $K1$ and $K2$. The relevant call to *Expand* creates an OR node over ASG nodes, each of which contains an assignment to the IFVs, so that Defs. 5.14 and 5.15 force the cut set of K to belong to one of the disjuncts. An argument like the one for parallel composition now shows that all paths of the CPS of K descend into the subtrees for $K1$ and/or $K2$ and that the IVs of the two components are assigned to in the CPS. Noting that all the cut paths must ascend through the relevant ASG node, ensures that the CPS includes the IFV assignments too, as required. □

If we supplement the assignments in a cut path set with an assignment to true of all system FVs occurring in the cut set and an assignment to false of all system FVs not occurring in the cut set, we arrive at the following *correctness* statement, which says that whenever our abstract procedure produces an answer, it is a *true* answer, i.e. the procedure does not lie.

Proposition 5.19 (Correctness). Let S be (the I/O transformer of) a system, let $FC \subseteq FV(S)$ be a fault configuration, and TLE a top level event. Let $RT = GenRT(TLE, S)$. If $FC \in CCS(RT)$, then FC is a cut set for TLE and S.

Proof. By tracing the assignments made during the proof of the previous proposition and the remark following it, it is clear that they amount to an assignment that satisfies (44). □

If a cut set is empty, the system is capable of producing the TLE in a fault-free manner, whereas if a cut set is nonempty, faults are involved. The next proposition confirms that this split corresponds to the output/concedes split of retrenchment.

Proposition 5.20. Let S be a system, TLE a top level event for S, and $FC \subseteq FV(S)$ a cut set for TLE and S. If $FC = \varnothing$ then O_S is satisfiable. If $FC \neq \varnothing$ then C_S is satisfiable.

Proof. A simple induction on system structure. For the BC case, $FC = \varnothing$ corresponds to $A.F = \mathsf{false}$ and the satisfiability of the left disjunct of (41), hence of O_S. The only other possibility, $FC = \{A.F\}$, corresponds to $A.F = \mathsf{true}$ and the satisfiability of the right disjunct of (41), hence of C_S. For the inductive step it is enough to partition the FC according to the disposition of the BCs in the two components, and to note that a compound O requires both component Os to be true, while a compound C requires at least one component C to be true. □

Proposition 5.21 (Completeness). Let S be (the I/O transformer of) a system, let $FC \subseteq FV(S)$ be a fault configuration, and TLE a top level event. Let $RT = GenRT(TLE, S)$ and let FC be a cut set for S and TLE. Then $FC \in CCS(RT)$.

Proof. Suppose $FC = \varnothing$. Then since FC is a cut set for S and TLE, by Prop. 5.20, we know that O_S must be satisfiable. Therefore case (b) or (d) of Fig. 9 applies, and *GenRT* creates a GOAS node. This contains a set of assignments to all non-fault variables, which generates FC by Defs. 5.14 and 5.15.

Otherwise $FC \neq \varnothing$, and by Prop. 5.20, C_S is satisfiable. So case (c) or (d) of Fig. 9 applies, and *GenRT* creates an $n : C$ node and calls $Expand(n)$. We proceed by induction on the truncated parse tree of S as used in Prop. 5.18, with the following inductive hypothesis:

> Let K be (the I/O transformer of) a system for which C_K holds, $\varnothing \neq FC \subseteq FV(K)$ a fault configuration, and Φ an assignment to all XVs of K. If there are j and p such that:
>
> $$K(j,p) \wedge \Phi \wedge \bigwedge_{F \in FC} F \wedge \bigwedge_{G \in \overline{FC}} \neg G \tag{45}$$
>
> holds, then $FC \in CCS(N)$, where N is the tree produced by the topmost call to *Expand* in the application of *GenRT* to Φ and K.

It is easy to see that the inductive hypothesis implies what is required. We let $K = S$ and Φ be the assignments to TLXVs in one of the ASG nodes created by the call to *GenRT*. Since such an assignment makes (44) true, we deduce the truth of (45), the assumption of the hypothesis. Hence the conclusion of the hypothesis holds, i.e. that $FC \in CCS(N)$ where N is the tree produced by the topmost call to $Expand(n)$. Noting that for the C_S satisfiable case, $RT = GenRT(TLE, S)$ differs from N only by the addition of an OR structure connecting N to the TLE node, $FC \in CCS(N)$ and Defs. 5.14 and 5.15 immediately imply $FC \in CCS(RT)$.

We now establish the induction.

Base cases: Firstly, $K \equiv A$, with A a faulty BC. We must have $FC = \{A.F\}$. We know that there exist j and p such that $A(j,p) \wedge \Phi \wedge A.F$ holds. Also, by (41) we have that $A(j,p) \wedge \Phi \wedge A.F$ is equivalent to $\Phi \wedge A.F \wedge C_A$. It follows that $\Phi \wedge C_A$ is consistent. Therefore the call $Expand(n)$ in $GenRT(\Phi, K)$ creates N, which consists of an ASG node with a BF child containing $A.F$. By Defs. 5.14 and 5.15, $FC = \{A.F\} \in CCS(N)$, as required.

Secondly, K is a non-faulty subsystem. The inductive hypothesis holds automatically since its assumption is false.

Inductive cases: $K \equiv K1 \circ K2$, where we have two sub-cases, $K \equiv K1 \fatsemi K2$ and $K \equiv K1 | K2$. We prove the first case in the simple version where the outputs of K1 exactly match the inputs of $K2$ (the more general version merely adds notational clutter). The second case is similar and is left to the reader.

Let j and c be the input and output variables of $K1$, and c and p be the input and output variables of $K2$, so that the IFVs of the composition are c.

Now the inductive hypothesis is an implication, and so is only nontrivial if its assumptions hold. To prove it, we

Function *RTtoFT*(*root*)
Input: A RT rooted at *root* produced by *GenRT*
Output: A FT for *TLE* and *S* given by *root*
Begin
1 **If** The RT does not contain an OR node
2 **Then return** *root*
3 **Else** Short-circuit all OAS nodes (i.e. connect the child node to the parent node and discard the OAS node).
4 Short-circuit all OR nodes with just one child node.
5 **While** There is an ASG node with another ASG node as child
6 **Do** Conjoin the assignment in the child ASG into the parent ASG node and short-circuit the child ASG node.
7 **EndDo**
8 **If** There is an AND or OR descendant of *root*
9 **Then** *Check*(*n*) **where** *n* is the nearest AND or OR descendant of *root*.
10 **EndIf**
11 **return** *root*
12 **EndIf**
End

Fig. 12. The RT to FT conversion algorithm.

must derive the conclusions of the implication on the basis that these assumptions are true, namely that there are values for $j, c, p,$ for which:

$$K1(j,c) \wedge K2(c,p) \wedge \Phi \wedge \bigwedge_{F \in FC} F \wedge \bigwedge_{G \in \overline{FC}} \neg G \tag{46}$$

hold.

Let m be values for c which make (46) true. Then

$$K1(j,c) \wedge K2(c,p) \wedge \Phi \wedge \phi_K \wedge \bigwedge_{F \in FC} F \wedge \bigwedge_{G \in \overline{FC}} \neg G \tag{47}$$

is satisfiable, where $\phi_K \equiv \bigwedge_c (c = m) \in VS(D,c)$.

From (47) we can derive the truth of both:

$$K1(j,c) \wedge \Phi_1 \wedge \phi_K \wedge \bigwedge_{F \in FC_1} F \wedge \bigwedge_{G \in \overline{FC_1}} \neg G \tag{48}$$

where $FC_1 = FC \cap FV(K_1) \subseteq FV(K_1)$, and Φ_1 is Φ with all assignments to variables not belonging to $K1$ erased (recalling that Φ is an assignment to all XVs of K); and:

$$K2(c,p) \wedge \Phi_2 \wedge \phi_K \wedge \bigwedge_{F \in FC_2} F \wedge \bigwedge_{G \in \overline{FC_2}} \neg G \tag{49}$$

where $FC_2 = FC \cap FV(K_2) \subseteq FV(K_2)$, and Φ_2 is Φ with all assignments to variables not belonging to $K2$ erased. Since Φ_1 contains assignments to at least j and Φ_2 contains assignments to at least p, neither Φ_1 nor Φ_2 is trivial. It is now easy to see that $\Phi_1 \wedge \phi_K$ is an assignment to all XVs of $K1$ and that $\Phi_2 \wedge \phi_K$ is an assignment to all XVs of $K2$. Evidently also $FC_1 \cup FC_2 = FC$ and $FC_1 \cap FC_2 = \varnothing$.

We can now apply the inductive hypotheses to $K1$ and $K2$, noting that at least one of C_{K1} or C_{K2} will be valid. Suppose O_{K1} holds in which case there is nothing to prove for it, and then C_{K2} must hold. In that case, by hypothesis, the call to *Expand* for $K2$ produces a tree $N2$ such that FC_2 is in $CCS(N2)$. Observing that no FV for any BC in $K1$ is true, we see that the conjunct $\bigwedge_{F \in FC_1} F$ in (48) is trivial, and that the call to *Expand* for K produces a tree N with an OR root, below which is an ASG node containing $\phi_K \equiv \bigwedge_c (c = m) \in VS(D,c)$, below which is another OR, for which the relevant branch will contain an OAS node with subtree N_2 descending from it (see Fig. 11.(c)). Since the top node of N is an OR and the node below the ASG is also an OR, $CCS(N)$ will contain $CCS(N2)$ as a subset, giving us the required conclusion in this case. The argument for O_{K2} and C_{K1} holding is similar.

If both C_{K1} and C_{K2} hold, instead of an OAS with descending subtree, *Expand* creates an AND node with subtrees $N1$ and $N2$ for $K1$ and $K2$ respectively. Noting that FC_1 and FC_2 partition FC, that $FC_1 \in CCS(N1)$ and $FC_2 \in CCS(N2)$ by the inductive hypotheses, and noting the rule for AND in the *CCNS* function, we derive the required conclusion. □

5.3. Turning Resolution Trees into Fault Trees

The preceding section proved the crucial soundness and completeness results for the RT algorithm. Once we have the resolution tree, we can potentially do many things with it. In this paper we illustrate this potential by describing how to extract a fault tree of a relatively conventional form from it in a mechanical manner.

Fig. 12 gives an algorithm, *RTtoFT*, which performs the required task. *RTtoFT* calls two further functions, *Check* and *InsertIE* to complete its work. These are given in Fig. 13. We briefly summarise the properties of this transformation.

Definition 5.22 (Fault Tree). A Fault Tree is a tree such that for every path from the root to a leaf, the node types encountered form a sentence describable by the regular expression:

$$\mathrm{FT} \equiv \mathrm{TLE}\ (\ (\mathrm{AND} + \mathrm{OR})\ \mathrm{IE}\ )^*\ \mathrm{BF} \tag{50}$$

The standard definition [VSD$^+$02] insists on intermediate events (i.e. IE nodes) interleaving between distinct occurrences of logical connectives, and this is reflected in (50). Note that (50) only partially defines FTs since it must be supplemented with information on node arities etc. in order to pin down the tree structure precisely.

In this paper we will modify the permitted possibilities in a number of ways. Firstly we will allow ASG nodes (or amalgamations thereof) to take the place of IE nodes when they have already been planted in a suitable position by *GenRT*. This is a natural choice given that, as mentioned in Section 5.2, such nodes correspond to genuine intermediate events in the fault tree. Secondly, we will allow the top levels of the FT to feature a GOAS node if non-faulty behaviour can give rise to the TLE. Thirdly, we allow removal of OAS nodes; an alternative, described in Section 5.2, could be to insert them as external events, but the current choice is sufficient for our purposes. The remaining modifications, such as inserting IE nodes between consecutive connective nodes, are mostly needed for housekeeping, namely to conform to the standard fault tree notation. Accordingly, we give the following definition. Again, the specification is only partial, but it will be sufficient for our purposes since we already have the RT to start with.

Definition 5.23 (Modified Fault Tree). A Modified Fault Tree is a tree such that for every path from the root to a leaf, the node types encountered form a sentence describable by the regular expression:

$$\mathrm{FT} \equiv \mathrm{TLE}\ [\ [\mathrm{OR}]\ \mathrm{GOAS}\ ] \quad + \quad \mathrm{TLE}\ [\mathrm{ASG}]\ (\ (\mathrm{AND} + \mathrm{OR})\ (\mathrm{ASG} + \mathrm{IE})\ )^*\ \mathrm{BF} \tag{51}$$

Proposition 5.24 (*RTtoFT* Properties). The algorithm *RTtoFT* converts the RT produced by *GenRT* into a FT according to Def. 5.23.

Proof sketch. It is not hard to see that the output of *GenRT* is a tree whose root-to-leaf paths consist of:

1. a path through one of the top level cases in Fig. 9.(a)-(d),
2. zero or more occurrences of root-to-leaf paths through Fig. 11.(b) or Fig. 11.(c) with its root overwriting the *C* leaf node of the path-so-far (provided the leaf node is indeed a *C* node),
3. an occurrence of Fig. 11.(a), with its root overwriting the *C* leaf node of the path-so-far (provided the leaf node is indeed a *C* node).

Algorithm *RTtoFT* starts by eliminating OAS and one-child OR nodes, and then amalgamating chains of ASG nodes. After this, the only possible deviation of a path in the tree from the form in (51) is the presence of consecutive AND or OR nodes in the path. To change these paths into legal ones, we need to interleave IE nodes between consecutive connective nodes; this is the job of the functions *Check* and *InsertIE* of Fig. 13, called at the top level in line 9 of Fig. 12. It is not hard to see that these routines do the job required. □

We can prove that the fault trees generated according to the algorithm in Fig. 12 enjoy the following restricted notion of causality, which is related to the structural decomposition of the system (compare also the discussion in Section 3).

Proposition 5.25 (Causality). Provided substructures closest to the top level inputs are nested more deeply than ones closer to the top level outputs, then the FT algorithm generates a 'causal' FT, in the sense that the variables closest to the inputs are assigned to in ASG and IE nodes that occur deeper in the FT than the ASG and IE nodes for variables closer to the outputs.

Proof. Since the RT algorithm works top-down, if the *K* node for $\frac{o}{91}$ occurs higher in the parse tree than the *K* node for $\frac{o}{92}$, then the RT algorithm plants the ASG nodes for $\frac{o}{91}$ higher in the RT than the ASG nodes for $\frac{o}{92}$. The *RTtoFT* algorithm preserves this ordering. Therefore to achieve a 'causal' FT, in which the ASG nodes encountered along a

Function *Check*(n)
Input: A node n of a pre-processed RT
Begin
1 **ForAll** Children n_k of n such that n_k is an AND or OR node
2 **Do** *Check*(n_k)
3 *InsertIE*(n_k)
4 **EndForAll**
5 **ForAll** Children n_k of n such that n_k is an ASG node
6 **Do** *Check*(n_{k_1}) **where** n_{k_1} is the child of n_k
7 **EndForAll**
8 **return**
End

Function *InsertIE*(n)
Input: A node n of a pre-processed RT
Begin
1 Create an IE node containing the conjunction of the formulae in the ASG or IE children of n if n is an AND node,
or containing the disjunction of the formulae in the ASG or IE children of n if n is an OR node.
2 Interpose the IE node between n and its parent (i.e. make the IE node the child of n's parent, and make n the IE node's child).
3 **return**
End

Fig. 13. The *Check* and *InsertIE* functions.

path through the FT toward the root assign to variables in an order not contrary to the variables encountered along the dataflow, it is sufficient to ensure that earlier sequential compositions (according to dataflow ordering) are more deeply nested than later sequential compositions. N.B. Since all systems are finite and acyclic, such a nesting can always be found. □

The astute reader will notice that the tree in Fig. 7 features variable assignment $c7 = 0$ below variable assignment $c5 = 0$, despite the fact that $c5$ causally precedes $c7$ in the structure $K0$-$K4$ given in Fig. 5. There is no contradiction with Prop. 5.25, since Prop. 5.25 makes essential use of angelic nondeterminism, whereas the algorithm sketch that derives Figs. 6 and 7 mirrors a practical algorithm, that perforce avoids angelic nondeterminism, and just follows the data dependencies. One fact is clear though. Such anomalous cases can only arise for assignments which have no dataflow to the TLE, since if there *were* any dataflow to the TLE, a derivation following a properly constituted input-innermost structure, would encounter the need to assign to $c7$ before encountering $c5$. Minimisation, discussed next, in which such non-needed derivations are discarded, prunes the relevant subtrees anyway.

6. Structured Minimisation

In practical fault analysis, it is of particular interest to generate the minimal fault configurations, the so-called *minimal cut sets* (MCSs for short), consisting of the fewest possible basic faults that cause a particular TLE. This is because including redundant fault configurations may lead to an unacceptable combinatorial explosion of causes for a typical TLE encountered in practice. Furthermore, minimal cut sets are often used as the basis for quantitative evaluation of fault trees.

The traditional technique for discovering MCSs is subsumption. In principle, one needs to generate all possible configurations that cause a fault (which may lead to the combinatorial explosion already noted), and then check them against one another: any that are subsumed by simpler configurations are discarded. Obviously 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 [CM92, CM93, Rau93, RD97] based on binary decision diagrams (BDDs) [Bry92] 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 techniques described in this section can be used to generate a minimised tree in parallel with (or when only minimal cut sets are required, in place of) the construction of the main resolution tree, described in Section 5. The advantage of such techniques is to avoid a brute-force subsumption on the flat representation of minimal cut sets. Similar ideas may also offer opportunities for optimizing the computation of minimal cut sets in the symbolic realm of FSAP. This is briefly discussed in Section 8.

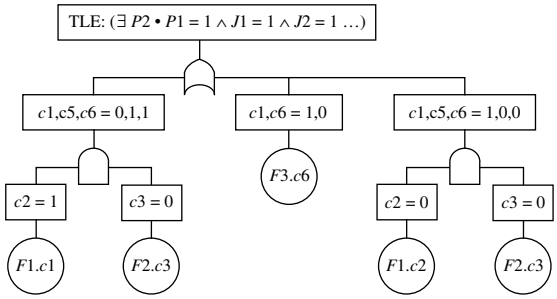

Fig. 14. A Minimised Fault Tree for the TLE of *Fred*.

The construction described in this section builds upon some minimisation opportunities and rules, which are illustrated below on our running example.

M.1: Discarding non-needed subtrees. If, during the FT construction, an OR choice arises which entails a fault which leads to an assignment to some variable whose value does not affect the validity of the TLE (e.g. there is no dataflow from the fault to the TLE), then the subtree rooted at this OR choice can be discarded immediately since the TLE is satisfied without it. In general, we call such faults incidental faults. An example is the subtree of Fig. 6, involving $F3.c7, F1.c2, F2.c3$, since there is no dataflow from $c7$ to the TLE. As in this 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 subtrees at input-insensitive faults. If, during the FT construction, a fault is generated which is independent of any input to the component in question, the subtree corresponding to the elaboration of those input values can be discarded immediately. An example occurs in Fig. 6, in case **TLE.R**, which considers $O_{K3}^{9}C_{F3,c6} \vee C_{K3}^{9}O_{F3} \vee C_{K3}^{9}C_{F3,c6}$. Since $F3.c6$ is a *stuck_at_zero* fault, when it is true, the behaviour of $K3$ is immaterial. So we can immediately discard the term $C_{K3}^{9}C_{F3,c6}$ in favour of $O_{K3}^{9}C_{F3,c6}$, and indeed we need not even explore the satisfiability of O_{K3} in detail either.

M.3: 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. E.g. in Fig. 6, $F1.c2$ subsumes $F1.c2 \wedge F2.c3$.

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.

Note that it could be said that all of **M.1**-**M.4** are instances of subsumption, if one understands subsumption from a semantic perspective, i.e. formulae are viewed up to logical equivalence. Unfortunately, algorithms running on real computing machinery cannot view arbitrary formulae up to logical equivalence without investing the effort to prove the logical equivalence, and it is this work that we are trying to save. Thus it could be said that **M.1** concerns the subsumption of $C_{K3}^{9}C_{F3,c7}$ by $C_{K3}^{9}O_{F3}$. However this subsumption only becomes explicit when both terms are reduced to the cut sets that give rise to their respective behaviours, at which point a machine can indeed determine that $\{F1.c2, F2.c3\}$ is a subset of $\{F1.c2, F2.c3, F3.c7\}$ and thus subsumes it. By noting the circumstances during the FT algorithm itself, the reduction to cut sets of $C_{K3}^{9}C_{F3,c7}$ is avoided. In **M.2** a similar thing occurs. Thus it could be said that $C_{K3}^{9}C_{F3,c6}$ is subsumed by $O_{K3}^{9}C_{F3,c6}$ on the grounds that the cut set of the latter, i.e. $\{F3.c6\}$, is a subset of all cut sets of the former. But again, this only becomes plain to a machine when all the cut sets have been computed. Bringing the suggested optimisation to bear, saves in this case, a considerable amount of work.

When we apply the above insights to the running example whose RT is indicated in Fig. 6, we get a considerably smaller tree. We transform this into a legal FT as per [VSD+02], containing just the MCSs, by the process outlined in Section 4. Doing this, we end up with the minimised fault tree in Fig. 14.

7. Formal Minimisation

In the following, we denote by $Min(RT)$ the resolution tree minimised according to the rules described in the previous section. We give the following definition.

Definition 7.1 (Subsumption). Let S be a system and TLE a top level event. Let CS_1 and CS_2 be two cut sets for TLE and S. We say that CS_1 subsumes CS_2, written $CS_1 \sqsubseteq CS_2$, iff $CS_1 \subseteq CS_2$.

We have the following obvious results.

Proposition 7.2 (Correctness). Let S be a system and TLE a top level event. Let $RT = GenRT(TLE, S)$. Then the minimized resolution tree is correct, that is, the following holds.

$$\bigvee_{CS \in CCS(RT)} \left(\bigwedge_{F \in CS} F \right) \Leftrightarrow \bigvee_{CS \in CCS(Min(RT))} \left(\bigwedge_{F \in CS} F \right) \tag{52}$$

Proposition 7.3 (Minimality). Let S be a system and TLE a top level event. Let $RT = GenRT(TLE, S)$. Then the cut sets in $CCS(Min(RT))$ are minimal, that is, for every CS_1, $CS_2 \in CCS(Min(RT))$ we have that $CS_1 \sqsubseteq CS_2 \Rightarrow CS_1 = CS_2$.

Beyond the preceding, is the issue of how minimisation can be incorporated into 'on-the-fly' FT generation. The material in Section 5 opens the way for this, though it has to be accepted, that being focused on describing an abstraction of a practical algorithm (moreover one that uses angelic features), it might be misleading in terms of the practicality of the minimisation opportunities it offers. Nevertheless, we have the following result.

Proposition 7.4 (*GenRT* Safe Optimisation). If only minimal cut sets are of interest, then in the *GenRT* algorithm of Section 5:

1. It is safe to dispense with case Fig. 9.(d) in favour of Fig. 9.(b).
2. In the *Expand* function, in cases Fig. 11.(b) or Fig. 11.(c), if at least one of the OAS/C branches is realisable, then it is safe to dispense with the corresponding $AND(C_1, C_2)$ subtree.

Proof. Regarding 1., if O_S is realisable, then it leads to the empty cut set which is obviously minimal, so there is no need to explore C_S. Regarding 2., for parallel composition, in case Fig. 11.(b), the two components are independent since the variables that remain to be instantiated at the point of creation of a Fig. 11.(b) subtree are just IVs of one or other component. Provided say $O_1 \wedge C_2$ is realisable, it will lead to computed cut sets that only have C_2 faults. Clearly any cut set of $C_1 \wedge C_2$ will include C_1 faults along with a set of C_2 faults, so will be subsumable by the C_2 subset, since these will be a cut set, by independence. The argument for sequential composition is the same, since the only variables shared by the two components, their IFVs, have already been assigned at the top of a Fig. 11.(c) subtree, thereby decoupling the two components. □

The last point in the preceding proof indicates why the optimisation of Prop. 7.4 is merely safe rather than optimal. In a sequential composition, there can be many assignments to the IFVs that realise the concession for the composite subsystem. There may be one assignment leading to a cut set $CS_{1'} \subseteq FV1$ involving the fault variables of component 1 alone, and another assignment leading to a cut set $CS_{12} \subseteq FV1 \cup FV2$, involving the fault variables of both. Now $CS_{1'}$ may be a subset of $CS_{12} \cap FV1$, in which case it subsumes CS_{12}; or the opposite may hold; or they may be incomparable. In the latter two cases CS_{12} may or may not itself also be minimal. Unfortunately, which of these cases applies, is not apparent at the point of generating a Fig. 11.(c) subtree, at least not without a substantial additional appeal to angelic nondeterminism.

Despite the above, a tightening of Prop. 7.4 can be made if there is enough determinism around.

Definition 7.5 (Definite TLE). A TLE for a system S is definite iff it is equivalent to an assignment to all TLXVs.

Proposition 7.6 (*GenRT* Deterministic Optimisation). Let S be a system, all of whose BCs are deterministic, and let TLE be a definite top level event for S. Then the optimisations of Prop. 7.4 ensure that *only* minimal cut sets are generated by *GenRT*.

Proof. The assumption of a definite TLE and of a deterministic S, ensure that provided $TLE \wedge S \wedge C_S$ is realisable, the TLE node has a unique ASG node in Fig. 9.(c), and each sequential composition generates a unique ASG node in Fig. 11.(c). Let CS_A and CS_B be two cut sets generated by the optimised *GenRT* and suppose that $CS_A \subseteq CS_B$. Let $B.F \in CS_B - CS_A$. Then in the truncated parse tree of S relating to the generation of CS_B, there will be a nearest

ancestor of BC B where the CS_A run of *GenRT* selected an OAS/C branch while the CS_B run selected the AND(C_1, C_2) subtree. By determinism, both are consistent with a single assignment to all variables other than IVs of the composite, which decouples the two components. If the CS_A run selected an OAS/C branch, then the OAS/C branch is realisable. Therefore the optimised *GenRT* would have discarded the AND(C_1, C_2) subtree in its favour, and we deduce that $B.F \notin CS_B - CS_A$ after all, giving $CS_B - CS_A = \varnothing$. Hence only minimal cut sets are generated. □

It is now interesting to compare the above results with what was done in Section 6. There, it was clear that **M.1**-**M.3** could be applied 'on-the-fly'. They are clearly special cases of the optimisations in Prop. 7.4, which thus subsumes them. However Prop. 7.4 relies in general on angelic nondeterminism, and in a practical setting, angelic nondeterminism costs — for instance with its more extravagant use, one could simply posit that the minimal cut sets were angelically returned. So in reality there will be a tradeoff between the cost of resolving any angelic nondeterminism in a minimal cut set algorithm, and the cost of mopping up residual non-minimal cut sets via a final brute-force subsumption check.

8. Related Work

The present work falls into the area of model based safety analysis [BV$^+$03b, BCC$^+$03, B$^+$06, BV07, BCT07]. Model based safety analysis is carried out on formally specified models which take into account system behaviour in the presence of malfunctions, that is, possible faults of some components. In particular, this paper builds upon previous work describing algorithms for automated fault tree generation, in particular those implemented in FSAP [BV07, BCT07, FSA], a platform for supporting the development and safety assessment of complex systems. Incorporation of the algorithms described in this paper in FSAP is work in progress - a more thorough comparison and discussion of some relevant issues is given in PaperII.

The FSAP platform has been developed within three European-Union-sponsored projects involving various research centers and industries from the avionics sector, namely the ESACS[10] (Enhanced Safety Assessment for Complex Systems), ISAAC[11] (Improvement of Safety Activities on Aeronautical Complex systems) and MISSA[12] (More Integrated Systems Safety Assessment) projects. For a more detailed description of the project goals we refer to [BV$^+$03b, BCC$^+$03, B$^+$06]. The FSAP platform has been used to carry out safety assessment of models at industrial level, see e.g. [BCC$^+$03].

Regarding model based safety analysis, we also mention [JH05, JMWH05], sharing some similarities with the ISAAC approach. In particular, the integration of traditional development activities with safety analysis activities, based on a formal model of the system and a clear separation between the nominal model and the fault model, are ideas that have been pioneered by ESACS [BV$^+$03b]. Finally, we mention [MTH03, TLM02, TM03], sharing with the ISAAC project the application field (i.e. avionics), and the use of NuSMV as a target verification language.

The retrenchment based algorithms described in this paper improve over the ones described in [BV07] for two reasons. First, they allow the generation of structured fault trees, which are more informative than the flat fault trees currently available in FSAP. Second, assuming the enhancements of PaperII, they allow the taking of dynamic information into account, e.g. they can deal with transient failures, and feedback. However they come at a price, that of potentially excessive complexity, so their realisation must be approached with caution.

We distinguish two different forms of complexity. The complexity of the analysis itself and of an implementation of it, and the complexity (in terms of, e.g., size and readability) of the generated results. As regards complexity of the analysis, we defer the discussion to PaperII [BB10], where we briefly discuss the main issues related to a practical implementation of our techniques using symbolic model checking, in the more general context of time-dependent circuits. Here it will suffice to say that holding all the details of the analysis in the symbolic world may be prohibitive, on both time and space grounds, and hence details must be introduced with care and in a controlled way. A full discussion of this topic, and a thorough outline of the similarities and differences between the retrenchment-based techniques and the symbolic ones is, however, outside the scope of these papers, and will be published elsewhere.

Concerning complexity of the generated results, there are two ways to deal with this issue, similarly to more traditional manual analysis techniques. First, it is possible to restrict the boundary of the analysis. For instance, it is possible to limit the analysis only to specific sub-systems or equipments of interest. Second, it is possible to reduce the level of resolution of the analysis, using, e.g. abstraction (and refinement) techniques. For instance, it is possible

[10] `http://www.esacs.org`.
[11] `http://www.isaac-fp6.org`.
[12] `http://www.missa-fp7.eu`.

to view the faults of a given major component as being elementary, without tracing their causes down to more basic components. This is not different from what happens in traditional fault tree analysis. Moreover, as discussed in Section 2.4, the retrenchment-based framework allows for progressive decomposition and recomposition of hierarchical models, based on the so-called Tower Pattern construction [BJ, Jes05, BPJS05, BPJS06a, BPJS06b], thus facilitating integration of results obtained at different levels of detail. Finally, the readability of the generated trees, e.g. as regards the readability of the automatically generated intermediate events, relies on human interpretation and postprocessing by safety analysts.

Our work has been inspired by Hip-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [Pap00, PM01, PMSH01], and the FoSaM (Formal Safety Model) approach described in [BV$^+$03b, BV07]. Hip-HOPS is a framework incorporating a mechanical fault tree synthesis algorithm based on the structure of the design model. The synthesis of the fault tree is based on an exploratory analysis called FFA (Functional Failure Analysis), used to identify the failure modes of a given component, and a tabular technique called IF-FMEA, whose aim is to generate a model of the local failure behaviour of the component under investigation. IF-FMEA tables are conceptually similar to the Failure Propagation and Transformation Notation [FMNP94], a notation used to represent the failure propagation in a system via a number of modules, each module corresponding to an abstraction of a set of fault trees describing a particular component. The focus of the Hip-HOPS methodology is not on generating fault trees *per se*, but in properly organizing such fault trees in accordance with the structure of the design model, and in managing model evolution. The result of Hip-HOPS is a hierarchical model that progressively records with increasing detail the implementation of the system. Our work is instead focused on the automation of the fault tree generation process, starting from a formal specification of both system and fault models. Furthermore, we discussed how the synthesis algorithm can be coupled with suitable on-the-fly tactics to perform local minimal cut set computation, reducing the overall computational effort.

A large amount of work has been done in the area of probabilistic safety assessment (PSA) and in particular on *dynamic reliability* [Siu94]. Dynamic reliability is concerned with extending the classical event or fault tree approaches to PSA by taking into consideration the mutual interactions between the hardware components of a plant and the physical evolution of its process variables [MZDL98]. For different approaches to dynamic reliability see e.g. [Ald87, Pap94, CIMP92, MZDL98, SD92]. These approaches are mostly concerned with the problem of fault tree evaluation, whereas our focus was on automatic synthesis. Also concerned with fault tree evaluation is DIFTree [MDCS98], a methodology for the analysis of dynamic fault trees, implemented in the Galileo tool [SDC99]. It uses a modularisation technique [DR96] 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 [AS98]. 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. Finally, we mention [Sch03, TS03], both concerned with automatically proving the consistency of fault trees using model checking techniques; [TS03] presents a fault tree semantics based on Clocked CTL (CCTL) and uses timed automata for system specification, whereas [Sch03] presents a fault tree semantics based on the Duration Calculus with Liveness (DCL) and uses Phase Automata as an operational model.

Regarding the algorithms for fault tree generation used in FSAP, the minimisation routines used to extract the set of minimal cut sets are based on classical procedures for minimisation of Boolean functions, specifically on the implicit-search procedure described in [CM92, CM93, Rau93, RD97], based on BDDs [Bry92] (see also [RPA08] for enhanced methods to convert fault trees into BDDs). Alternative explicit-search and satisfiability based techniques for computation of prime implicants are described, e.g. in [MOMS98]. In addition, important optimisations, that are tailored to the generation of minimal cut sets, have been recently implemented [BCT07] on top the main routines for minimising Boolean functions. Some of these optimisations resemble the minimisation opportunities described in Section 6. In particular, rules **M.1** and **M.2** are taken care of by a combination of the BDD package reduction rules and the DCOI (Dynamic Cone of Influence) construction of [BCT07], whereas rules **M.3** and **M.4** are taken care of by dynamic pruning [BCT07]. The rules of Section 6 may offer further opportunities for optimisations. A thorough discussion of this topic lies beyond the scope of this paper, and will be reported elsewhere.

9. Conclusions and Forward Look to PaperII

In the preceding sections, we set up our machinery for treating fault injection via retrenchment, and explored its properties. We introduced the relevant formalisms for systems and faults, and the relevant facts about retrenchment, especially concerning composition. We showed how the retrenchment concessions could be analysed to generate a

deeply nested resolution tree in a structured manner. This deeply nested resolution tree is the core output of our technique, and provides a jumping off point for what could potentially be many kinds of further processing and analysis. Because of the widely known nature of the fault tree concept, we post-processed our resolution tree into a fairly conventional fault tree format, this being of additional significance since fault trees are the input format for commercial RAMS tools, see e.g. [ISO]. We backed up the example led discussion with a thoroughgoing theoretical treatment. We examined minimisation, and showed that the structured analysis gave rise to opportunities for on-the-fly minimisation, which is always welcome in situations in which complexity considerations defeat conventional approaches, as they do for automated fault tree generation. Finally, we were able to position these minimisation opportunities within the theoretical framework developed.

From an engineering perspective, we believe that the running examples presented throughout the paper should have convinced the reader of the usefulness and potentialities of automatically generated fault trees. While it is true that such fault trees are not guaranteed to obey the notion of causality given in [VSD$^+$02] (compare also the discussion in Section 3), the fact that structural decomposition, as described in this paper, closely follows the system dataflow, suggests, that in many cases of interest, the fault tree interconnections may in fact correspond to genuine causal relationships. Under the hypotheses of Prop. 5.25, which is one of the main results of this paper, these interconnections are built according to the system hierarchy. Hence, we can evidently draw a parallel between our formal fault tree derivation and the structural decomposition prescribed by the 'state of system' rule of [VSD$^+$02]. The decomposition has a natural correspondence with the expansion rules used in the resolution tree algorithm (compare Fig. 10).

The same expansion rules, when applied to basic components, have a natural counterpart with the 'state of component' decomposition prescribed by [VSD$^+$02]. In principle, our routines can address primary, secondary, and command faults. Clearly, given that our framework relies on a formal model of the system, faults may be discovered to the extent that they have been covered in the formal model, which may be not completely straightforward for some types of faults. For instance, modeling secondary faults due to operation of a component in an environment for which it is not qualified, requires a formal model of the environment covering the environmental conditions of operation that must be part of the analysis. A similar line of reasoning holds for command faults due to improper operation or human error. Regarding interpretation of faults, and their classification into the previous categories, it is the responsibility of the safety engineer to analyze the different cases that formal derivation generates and assign them a correct semantical interpretation in terms of primary, secondary or command faults. Our claim is that the intermediate events generated by our routines, when analyzed by a safety engineer that has a good comprehension of the system at hand, are a good starting point for this post-interpretation, as they contain assignments to all the system variables of interest. Finally, we wish to remark on the important role that automated routines, coupled with on-the-fly minimisation, may have in the computation of minimal cut sets (and hence for quantitative evaluation) — such computations may be manually infeasible for many complex models encountered in practice.

From a more general development process perspective, we believe that the techniques presented in this paper should be used to complement, not necessarily replace, traditional techniques based on expert review and judgment. Traditional fault tree analysis is an exploratory technique supported by expert knowledge and comprehension of the system at hand, hence it is guided, but not limited by design and safety models, and automated techniques are in fact constrained in what they can accomplish by such models. Any deficiency or limitation in the design and fault models can get exposed in the generated results. However, this is not a limitation of our techniques as such, but rather is intrinsic to automated formal verification in general. As is usually the case, the results of formal verification are meaningful to the extent that they can be reviewed and interpreted using human expertise. Then again, on the positive side, automatic analysis helps reduce the effort and prevent human error, in particular in the most repetitive and mechanical parts of the analysis. A comparison between automatically generated results and manually generated ones may expose problems that have escaped manual analysis, and possibly trigger system redesign recommendations. Moreover, it can expose problems in design and safety models, which may be used elsewhere for different kinds of analysis, hence resulting in useful feedback for design and safety engineers.

The work described in this paper applies to acyclic combinational logic circuits, in which there is no time delay between the consumption of inputs and the production of outputs, and no state. In PaperII, we address these shortcomings. The need to deal with clocked circuits entails the use of I/O streams and state components. It turns out that these aspects complicate considerably the composition and retrenchment machinery used in this paper, so a large part of PaperII is concerned with elaborating those details. However, once the basic machinery is in place, system descriptions reduce once more to collections of variables, each taking its values in some finite set. From that point onwards, the theoretical basis is like the one of this paper, and the theoretical considerations can therefore follow those here rather closely. PaperII therefore relies quite heavily on reusing the facts established in the latter parts of this paper.

References

[Ald87] T. Aldemir. Computer-assisted Markov Failure Modeling of Process Control Systems. *IEEE Transactions on Reliability*, R-36:133–144, 1987.

[AS98] 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.

[B+06] M. Bozzano et al. ISAAC, a Framework for Integrated Safety Analysis of Functional, Geometrical and Human Aspects. In *Proc. European Congress on Embedded Real Time Software (ERTS 2006)*, 2006.

[BB10] R. Banach and M. Bozzano. The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment II: Clocked and Feedback Circuits, 2010. Submitted.

[BC04] R. Banach and R. Cross. Safety Requirements and Fault Trees using Retrenchment. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Computer Safety, Reliability and Security*, volume 3219 of *LNCS*, pages 210–223, Potsdam, Germany, 2004. Springer.

[BCC+03] 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.

[BCT07] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Proc. Symposium on Automated Technology for Verification and Analysis (ATVA 2007)*, pages 162–176, 2007.

[BJ] R. Banach and C. Jeske. Retrenchment and Refinement Interworking: the Tower Theorems. Submitted. See [Ret].

[BJP08] R. Banach, C. Jeske, and M. Poppleton. Composition Mechanisms for Retrenchment. *J. Log. Alg. Prog.*, 75:209–229, 2008.

[BP98] R. Banach and M. Poppleton. Retrenchment: An Engineering Variation on Refinement. *B'98: Recent Advances in the Development and Use of the B Method: Second International B Conference, Montpellier, France, LNCS*, 1393:129–147, 1998.

[BP03] R. Banach and M. Poppleton. Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study. *Requirements Engineering Journal*, 8:266–288, 2003.

[BPJS05] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Finite Sequence Numbers and the Tower Pattern. In J. Fitzgerald, I. Hayes, and Tarlecki A., editors, *Formal Methods 2005*, volume 3582 of *LNCS*, pages 382–398, Newcastle, UK, 2005. Springer.

[BPJS06a] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Finite Exception Logs, and Validating the Small. In M. Hinchey, editor, *Software Engineering Workshop 30*, pages 234–245, Layola College Graduate Center, Columbia, MD, 2006. IEEE.

[BPJS06b] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties. In T. Margaria and B. Steffen, editors, *2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 82–90, Paphos, Cyprus, 2006. IEEE.

[BPJS07] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and Theoretical Underpinnings of Retrenchment. *Sci. Comp. Prog.*, 67:301–329, 2007.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BV03a] M. Bozzano and A. Villafiorita. Integrating Fault Tree Analysis with Event Ordering Information. *Proc. ESREL 2003*, pages 247–254, 2003.

[BV+03b] 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.

[BV07] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.

[CIMP92] G. Cojazzi, J. M. Izquierdo, E. Meléndez, and M. S. Perea. The Reliability and Safety Assessment of Protection Systems by the Use of Dynamic Event Trees. The DYLAM-TRETA Package. In *Proc. XVIII Annual Meeting Spanish Nucl. Soc.*, 1992.

[CM92] 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.

[CM93] 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.

[DBB92] J. Dugan, S. Bavuso, and M. Boyd. Dynamic fault tree models for fault tolerant computer systems. *IEEE Transactions on Reliability*, 41(3):363–377, 1992.

[DR96] Y. Dutuit and A. Rauzy. A Linear-Time Algorithm to Find Modules in Fault Trees. *IEEE Transactions on Reliability*, 45(3):422–425, 1996.

[dRE98] W. P. de Roever and K. Engelhardt. *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press, 1998.

[FMNP94] P. Fenelon, J. A. McDermid, M. Nicholson, and D. J. Pumfrey. Towards Integrated Safety Analysis and Design. *Applied Computing Review*, 2(1):21–32, 1994.

[FSA] The FSAP/NuSMV-SA platform. `http://sra.itc.it/tools/FSAP`.

[Int96] SAE International. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[ISO] Isograph. `http://www.isograph-software.com`.

[Jes05] C. Jeske. *Algebraic Integration of Retrenchment and Refinement*. PhD thesis, University of Manchester, 2005.

[JH05] A. Joshi and M. P. E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In R. Winther, B.A. Gran, and G. Dahll, editors, *Proc. Conference on Computer Safety, Reliability and Security (SAFECOMP 2005)*, volume 3688 of *LNCS*, pages 122–135. Springer, 2005.

[JMWH05] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proc. AIAA / IEEE Digital Avionics Systems Conference (DASC 2005)*, 2005.

[MDCS98] 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.

[MOMS98] V. M. Manquinho, A. L. Oliveira, and J. P. Marques-Silva. Models and Algorithms for Computing Minimum-Size Prime Implicants. In *Proc. International Workshop on Boolean Problems (IWBP 1998)*, 1998.

[MTH03] S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the Shalls. In *Proc. Formal Methods Europe (FM 2003)*, volume 2805 of *LNCS*, pages 75–93. Springer, 2003.

[MZDL98] M. Marseguerra, E. Zio, J. Devooght, and P. E. Labeau. A Concept Paper on Dynamic Reliability via Monte Carlo Simulation. *Mathematics and Computers in Simulation*, 47:371–382, 1998.

[Pap94] I. A. Papazoglou. Markovian Reliability Analysis of Dynamic Systems. In T. Aldemir, N. O. Siu, A. Mosleh, P. C. Cacciabue, and B. G. Göktepe, editors, *Reliability and Safety Assessment of Dynamic Process Systems*, volume 120 of *NATO ASI Series F*, pages 24–43. Springer, 1994.

[Pap00] Y. Papadopoulos. *Safety-Directed System Monitoring Using Safety Cases*. PhD thesis, Department of Computer Science, University of York, 2000. Tech. Rep. YCST-2000-08.

[PM01] 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.

[PMSH01] 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.

[Rau93] A. Rauzy. New Algorithms for Fault Trees Analysis. *Reliability Engineering and System Safety*, 40(3):203–211, 1993.

[RD97] 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.

[Ret] Retrenchment Homepage. `http://www.cs.man.ac.uk/retrenchment`.

[RPA08] R. Remenyte-Prescott and J.D. Andrews. An enhanced component connection method for conversion of fault trees to binary decision diagrams. *Reliability Engineering and System Safety*, 93(10):1543–1550, 2008.

[Sch03] A. Schäfer. Combining Real-Time Model-Checking and Fault Tree Analysis. In *Proc. Formal Methods Europe (FM 2003)*, volume 2805 of *LNCS*, pages 522–541. Springer, 2003.

[SD92] C. Smidts and J. Devooght. Probabilistic Reactor Dynamics II. A Monte-Carlo Study of a Fast Reactor Transient. *Nuclear Science and Engineering*, 111(3):241–256, 1992.

[SDC99] 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.

[Siu94] N. O. Siu. Risk Assessment for Dynamic Systems: An Overview. *Reliability Engineering and System Safety*, 43:43–74, 1994.

[TLM02] A. C. Tribble, D. L. Lempia, and S. P. Miller. Software Safety Analysis of a Flight Guidance System. In *Proc. AIAA / IEEE Digital Avionics Systems Conference (DASC 2002)*, 2002.

[TM03] A. C. Tribble and S. P. Miller. Software Safety Analysis of a Flight Management System Vertical Navigation Function – A Status Report. In *Proc. AIAA / IEEE Digital Avionics Systems Conference (DASC 2003)*, 2003.

[TS03] A. Thums and G. Schellhorn. Model Checking FTA. In *Proc. Formal Methods Europe (FM 2003)*, volume 2805 of *LNCS*, pages 739–757. Springer, 2003.

[VGRH81] 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.

[VSD+02] W.E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback. Fault Tree Handbook with Aerospace Applications. Technical report, NASA, 2002.