Safety Requirements and Fault Trees using Retrenchment

R. Banach and R. Cross

Computer Science Department, Manchester University, Manchester, M13 9PL, UK
banach@cs.man.ac.uk, r.cross@cs.man.ac.u

Abstract. In the formal modelling of safety critical systems, an initial abstract model captures the ideal, fault free, conception of the system. Subsequently, this model is enriched with the detail required to deal with envisaged faults that the system is designed to be robust against, resulting in a concrete extended system model. Normally, conventional refinement cannot provide a formal account of the relationship between the two models. Retrenchment, a liberalisation of refinement introduced to address such situations, allows model evolution, and is deployed to provide a formal account of the fault injection process that yields the extended system model. The simulation relationship of retrenchment is used to derive fault trees for the faults introduced during the injection process. A two bit adder example drawn from the FSAP/NuSMV-SA safety analysis platform is used to illustrate the technique.

1 Introduction

Safety-critical systems are required to behave in a predictable and acceptable way in the face of failures which have been forseen and designed for during their development. Generally speaking, the behaviour in both desired and undesired (degraded) situations must be fully documented to provide a comprehensive system description. The process of creating such a comprehensive description starts by writing down an ideal, fault free, abstract model of the system. In this model, only the desired behaviour of the system is expressed. Once this initial model has been created, the elicitation of safety requirements yields a fresh set of criteria that the system must meet. The incorporation of these into the ideal model yields an 'extended system model', which encompasses all system behaviour in both desired and undesired circumstances.

One framework for carrying out this task is the FSAP/NuSMV-SA safety analysis platform [11]. This is a toolkit that allows the creation of the ideal model, and then permits the automatic injection of possible faults into the ideal model from pre-defined libraries of generic faults. In this way the extended system model is obtained. Verification techniques such as Fault Tree Analysis can then be used to establish the root cause of a given failure, and relate it to the ideal model in an informal manner. Bozzano et al., the authors of the FSAP/NuSMV-SA platform, justify the use of such informal techniques in relating ideal and failure-sensitive models by pointing out the deficiencies of formal approaches,

saying: "...even when formal methods are applied, the information linking the design and the safety assessment phases is often carried out informally".

The reason that this happens is not hard to see, and is due to the fact that the traditional formal relationship between an abstract and a more concrete model is refinement [27], [16], [15]. Unfortunately the formal proof obligations of refinement are quite stringent, and are parameterised only by the retrieve (or abstraction) relation between the state spaces of the two models. This restriction prevents the description via refinement of many of the kinds of modification or adaptation that arise naturally during system development, and often forces the abstract and concrete models to be much closer together than one might ideally like. The impact of this for using refinement in the case of safety critical development would entail committing, more or less at the outset, to an abstract model that already contains essentially all the desired fault tolerant behaviour, and this obviously flies in the face of the desire to create just the ideal model first, and only later to consider the incorporation of faults (and of the system's responses to them) via a separate safety requirements analysis phase.

In this paper, we show that retrenchment, a liberalisation of refinement which was introduced precisely to address the excessive stingency of the formal proof obligations of refinement in such situations, can fare better in the role of intermediary between the ideal and extended system models. Retrenchment achieves this by introducing extra data, the within, output and concedes relations, into the principal refinement proof obligation. These extra relations depend on the the two systems being discussed and introduce much greater parameterisation into the description of the relationship between them. So we gain much greater flexibility in capturing the desired relationship between models. We exploit this flexibility to give a credible formal account of the relationship between ideal and fault tolerant systems in the process of fault injection. Most importantly, such a formal account offers opportunities for formally tying together —via the bridge offered by retrenchment— other aspects of the safety critical development, hitherto done separately (even if formally) in the ideal and extended model worlds. We illustrate this potential for fault tree generation, and more briefly comment on model checking.

The rest of this paper is structured as follows: Section 2 describes the two-bit adder, the case study that will serve as a running example in this paper, and explains the way in which it will evolve to model implementation-level faults. Section 3 overviews refinement in a simple partial correctness formulation, and applies it to the two-bit adder to show the shortcomings of refinement in this context. Section 4 introduces retrenchment, and shows how it can be used to give a much improved account of fault injection for the two-bit adder scenario. Section 5 introduces the simulation relationship of retrenchment, and shows how it can be used to generate the fault tree for the two-bit adder via a resolution-like procedure. Section 6 comments briefly on a similar approach to model checking. Section 7 concludes, looking forward to more detailed expositions of the topics introduced briefly in this paper.

2 The Two-Bit Adder Example

In the rest of the paper, we will frame our discussions of relationships between models in terms of an abstract system Abs, and a concrete one Conc, which will be two adjacent systems in the development hierarchy. The abstract system Abs, will contain a number of operations Op_{A}, taken from its set of operations $\mathsf{Ops}_{\mathrm{A}}$. An abstract operation Op_{A} is a transition relation for which the individual transitions will be written:

$$Op_{\mathrm{A}}(u, i, u', o)$$

or more evocatively:

$$u \text{ -}(i, Op_{\mathrm{A}}, o)\text{-> } u'$$

where u, u' are elements of the state space U, and i and o are elements of the input space I_{Op} and the output space O_{Op} respectively. Below we will use the arrow format to fix the signature of an operation in terms of states and outputs, and will use the Op_{A} format when we simply wish to utilise the transition relation as a component of a more complex relational expression. It turns out that for the simple examples that appear in this paper, all variables are either read-only or write-only, so we will model them using input and output variables, finessing the absence of genuine updatable state by using one-element state spaces containing a single '$*$' value. Thus the typical abstract operation becomes $* \text{ -}(i, Op_{\mathrm{A}}, o)\text{-> } *$. In addition to all this the system is started in an initial state u' satisfying the predicate $Init_{\mathrm{A}}(u')$.

Corresponding concrete operations are defined in exactly the same way, except with operations Op_{C} taken from $\mathsf{Ops}_{\mathrm{C}}$, with state space $v \in \mathsf{V}$, input $j \in \mathsf{J}_{Op}$ and output $p \in \mathsf{P}_{Op}$. Decoration is used to distinguish between before and after states of variables, with an operation Op being identified as abstract or concrete by use of a subscript A or C as necessary. The initialisation predicate is $Init_{\mathrm{C}}(v')$.

The example that will be used to highlight the use of retrenchment is the two-bit adder adapted from [11]. From now on, unless stated otherwise, all I/O variables take values in $\{0, 1\}$ the usual one bit space. We follow the structure of [11] closely, and so the abstract two-bit adder $Adder_{\mathrm{A}}$ is specified in terms of a subsidiary operation Bit_{A}. In fact Bit_{A} merely copies its input to its output, acting as a placeholder for subsequent injection of faulty behaviour. Note the use of pairing to turn two individual inputs i_1 and i_2 to $Adder_{\mathrm{A}}$, into a single formal input (i_1, i_2).

$$* \text{ -}(i, Bit_{\mathrm{A}}, o)\text{-> } * \quad = \quad (o = i)$$

$$* \text{ -}((i_1, i_2), Adder_{\mathrm{A}}, o)\text{-> } * \quad = \quad (o = (i_1 + i_2 \bmod 2))$$

Following [11] again, these operations are wrapped up in an enclosing operation $Main_{\mathrm{A}}$ which takes its input values, feeds them into the adder, and extracts the

result.

$$
\begin{aligned}
&*\text{-}((r_1, r_2), Main_{\mathrm{A}}, adder)\text{->}* \quad = \\
&\quad (\exists\, b_1, b_2 \bullet Bit_{\mathrm{A}}(*, r_1, *, b_1) \wedge Bit_{\mathrm{A}}(*, r_2, *, b_2) \wedge \\
&\quad Adder_{\mathrm{A}}(*, (b_1, b_2), *, adder))
\end{aligned}
$$

The above abstract idealised model describes how the system should behave when it is functioning correctly. The next step is to specify how the system should behave in degraded situations (i.e. when a fault occurs in a part of the system). So safety requirements are added, through the process of fault injection, to create a concrete extended system model.

We will inject two types of fault into the above system: a corruption of the input bit of the *Bit* module, and a stuck-at-zero fault in the *Adder* module. These are incorporated into the two concrete module defnitions of Bit_{C} and $Adder_{\mathrm{C}}$.

$$
\begin{aligned}
&*\text{-}(j, Bit_{\mathrm{C}}, p)\text{->}* \quad = \\
&\quad (\mathrm{ft} = no\_failure \wedge p = j) \vee \\
&\quad (\mathrm{ft} = inverted \wedge p = \neg j)
\end{aligned}
$$

$$
\begin{aligned}
&*\text{-}((j_1, j_2), Adder_{\mathrm{C}}, p)\text{->}* \quad = \\
&\quad (\mathrm{ft_a} = no\_failure \wedge p = (j_1 + j_2 \bmod 2)) \vee \\
&\quad (\mathrm{ft_a} = stuck\_at\_zero \wedge p = 0)
\end{aligned}
$$

Both the concrete *Bit* and *Adder* operations have an additional free variable, ft (fault_type), that indicates whether the fault in question is active. In the case of a faulty *Bit*, the input value is inverted, while in the case of a faulty *Adder*, the output is set to zero irrespective of the inputs. These definitions lead to a corresponding concrete extended *Main* operation, in which we relabel the (r_1, r_2) inputs and *adder* output as (s_1, s_2) and *sum* for later convenience.

$$
\begin{aligned}
&*\text{-}((s_1, s_2), Main_{\mathrm{C}}, sum)\text{->}* \quad = \\
&\quad (\exists\, c_1, c_2 \bullet Bit_{\mathrm{C}}(*, s_1, *, c_1) \wedge Bit_{\mathrm{C}}(*, s_2, *, c_2) \wedge \\
&\quad Adder_{\mathrm{C}}(*, (c_1, c_2), *, sum))
\end{aligned}
$$

Note that the choice of simply making the ft variables free in the definitions of Bit_{C}, $Adder_{\mathrm{C}}$ and $Main_{\mathrm{C}}$ is but one way of handling the relevant information. For us it was dictated by the desire for subsequent technical simplicity.

3 Refinement

In this section we outline a simple partial correctness formulation of model oriented refinement, and show that it will not describe the relationship between the abstract and concrete systems of the two bit adder. See [27], [15], [16] for treatments of model oriented refinement in general and [17], [23], [26], [28], for more details of the Z methodology, or [1], [24], [24] [25], for more details on B; both Z and B being particular incarnations of the model oriented approach.

For a concrete model of the kind we have been discussing to be a refinement of an abstract one, we need three things. Firstly that the sets of operation names at the two levels are identical, i.e. $\mathsf{Ops}_{\mathrm{A}} = \mathsf{Ops}_{\mathrm{C}}$. Secondly that the initilisation proof obligation (PO) holds:

$$Init_{\mathrm{C}}(v') \;\Rightarrow\; (\exists\, u' \bullet Init_{\mathrm{A}}(u') \wedge G(u', v'))$$

Thirdly for each corresponding pair Op_{A} and Op_{C}, the operation PO holds:

$$G(u, v) \wedge Op_{\mathrm{C}}(v, j, v', p)$$
$$\Rightarrow\; (\exists\, u', i, o \bullet Op_{\mathrm{A}}(u, i, u', o) \wedge G(u', v') \wedge (i = j) \wedge (o = p))$$

This shows that inputs and outputs are not permitted to change in the passage from abstract to concrete (in line with refinement's original objective of providing an implementaiton level model substitutable for all occurrences of the abstract one), and that the only scope for adjusting the relationship between the two models is via the retrieve relation G. Since G appears in both the antecedent and the consequent of the PO, there is often very little leeway indeed for the choice of G.

Let us examine the prospects for treating our running example via refinement. The initialisation PO is trivially satisfied because there is no state. On the other hand the operation PO for *Bit* obviously cannot be satisfied since equality of outputs does not always hold. Even if we are more permissive about I/O in the consequent of the PO, and supplant $(i = j) \wedge (o = p)$ by $In_{Op}(i, j) \wedge Out_{Op}(o, p)$ where $In_{Op}(i, j)$ and $Out_{Op}(o, p)$ are more general input and output relations as happens in eg. I/O refinement, [9], we are still in trouble since the only relation $Out_{Op}(o, p)$ that is independent of the fault_type parameter ft, and relates all instances of output pairs needed by the *Bit* operation, is the universal relation true, since when ft $= no\_failure$ we want $Out_{Op}(o, p)$ to be equality, whereas if ft $= inverted$ we want $Out_{Op}(o, p)$ to be inequality. This makes $Out_{Op}(o, p)$ completely uninformative, and in the absence of state, leaves the relationship between abstract and concrete completely unconstrained. Entirely analogous remarks apply in the case of the *Adder* operation.

Thus although refinement does a good job of keeping development steps in check, to the extent that behaviour defined at the abstract level is preserved at the concrete level, it proves very inflexible in formally capturing many kinds of development step that do not fall within its rather exacting constraints, but that are entirely justifiable on engineering grounds. See [18], [10], [19], [4] for more discussion of this and related points.

4 Retrenchment

Retrenchment ([3], [4], [5], [6], [7], [8], [2]), was introduced to provide a formal vehicle which, while resembling refinement up to a point, provides greater flexibility to express relationships between the models that designers genuinely have in mind during development. Retrenchment achieves this by parameterising the

relationship between abstract and concrete model in a development step more richly than by just a retrieve relation, having also within, output and concedes relations. These modifiy the shape of the operation PO as follows (the initialisation PO remains as for refinement):

$$
\begin{array}{l}
G(u,v) \wedge P_{Op}(i,j,u,v) \wedge Op_C(v,j,v',p) \\
\Rightarrow \ (\exists\, u', o \bullet Op_A(u,i,u',o) \wedge ((G(u',v') \wedge O_{Op}(o,p;u',v',u,v,i,j)) \\
\qquad \vee\ C_{Op}(u',v',o,p;u,v,i,j)))
\end{array}
$$

In the preceding, $P_{Op}(i,j,u,v)$ is the within relation and serves to constrain the impact of the implicational relationship between abstract and concrete models where this is desirable; also it generalises the $(i = j)$ input relation of refinement and permits the mixing of input and before-state information where this is considered appropriate. Likewise the output relation $O_{Op}(o,p;u',v',u,v,i,j)$ generalises the $(o = p)$ of refinement, and the presence of the other variables u', v', u, v, i, j allows the inclusion of any other facts that designers wish to highlight concerning the case where the abstract and concrete steps re-establish the retrieve relation $G(u',v')$. The crucial feature of retrenchment though, is the presence of the disjunction between $(G' \wedge O_{Op})$ and C_{Op}, where C_{Op} is the concedes relation. C_{Op} features the same variables as O_{Op}, giving the same level of expressivity as O_{Op}, but this time in circumstances where the retrieve relation $G(u',v')$ need not be re-established. This *weakening* of the postcondition gives the much greater flexibility alluded to earlier, but of course comes at a price: the price of the reworking of the whole of refinement theory. Refinement starts with the general principle of substitutability of abstract by concrete and *derives* its operation PO; retrenchment starts with the above operation PO and *derives* whatever general principles might survive the modification. (See the cited references for indication of progress on general principles).

In practical terms what retrenchment does is to honestly but formally allow an abstract model to evolve to a more realistic one, acting as a more usable specification constructor than refinement alone.[1] In the case of safety critical design, it is the desire to separate initial design from safety analysis (and not the demands of refinement) that drives the order in which features and requirements are incorporated into the model, and retrenchment is better able to accomodate this agenda than refinement, as we see when we revisit our case-study.

Starting with the *Bit* operation, it is not hard to see that the abstract and concrete versions can be conveniently related by a retrenchment. Here is a reasonable choice for the various component relations needed for the *Bit* operation:

$$G_{Bit}(*,*) \ = \ \mathsf{true}$$

[1] The usual assumption is that there is a process of gradually incorporating requirements via a collection of incomplete models (related to each other by retrenchment for example), until a final 'contracted model' is arrived at, which expresses all the needed requirements, and which can then be refined to an implementation. Such a picture is typically an oversimplification of a real development, but a nevertheless a useful one.

$$
\begin{aligned}
&P_{Bit}(i, j, *, *) \;=\; (i = j) \\
&O_{Bit}(o, p; *, *, *, *, i, j) \;=\; (\mathrm{ft} = no\_failure \wedge o = p) \\
&C_{Bit}(*, *, o, p; *, *, i, j) \;=\; (\mathrm{ft} = inverted \wedge o = \neg p)
\end{aligned}
$$

Note that a considerable element of choice has been exercised in designing the above relations. For instance, since there is no state to speak of, so that only inputs and outputs need to be kept under control, we could have chosen to put all the facts contained in the above O_{Bit} and C_{Bit} into either one of these relations, defaulting its counterpart to true or false as appropriate. The choice we actually made reflects our informal perception of which aspects of *Bit*'s behaviour are viewed as refinement-like, and which are regarded as degraded. For the given G, P, O, C, when we substitute the various components into the generic PO we get:

$$
\begin{aligned}
&\mathsf{true} \wedge (i = j) \wedge ((\mathrm{ft} = no\_failure \wedge p = j) \vee (\mathrm{ft} = inverted \wedge p = \neg j)) \\
&\Rightarrow\; (\exists\, *, o \bullet (o = i) \wedge \\
&\qquad ((\mathsf{true} \wedge \mathrm{ft} = no\_failure \wedge o = p) \vee (\mathrm{ft} = inverted \wedge o = \neg p)))
\end{aligned}
$$

which is more or less selfevident.

Moving to the *Adder* operation, the relations for the retrenchment can be chosen as follows:

$$
\begin{aligned}
&G_{Adder}(*, *) \;=\; \mathsf{true} \\
&P_{Adder}((i_1, i_2), (j_1, i_2), *, *) \;=\; \mathsf{true} \\
&O_{Adder}(o, p; *, *, *, *, (i_1, i_2), (j_1, j_2)) \;= \\
&\qquad ((\mathrm{ft_a} = no\_failure \wedge (i_1 + j_1 + i_2 + j_2 = 0 \bmod 2) \wedge o = p) \vee \\
&\qquad (\mathrm{ft_a} = no\_failure \wedge (i_1 + j_1 + i_2 + j_2 = 1 \bmod 2) \wedge o = \neg p)) \\
&C_{Adder}(*, *, o, p; *, *, (i_1, i_2), (j_1, j_2)) \;= \\
&\qquad (\mathrm{ft_a} = stuck\_at\_zero \wedge o = (i_1 + i_2 \bmod 2) \wedge p = 0)
\end{aligned}
$$

Note that the choice of within relation as true, is dictated by subsequent considerations. The reader can check that the operation PO now reduces to the easily verified:

$$
\begin{aligned}
&\mathsf{true} \wedge \mathsf{true} \wedge ((\mathrm{ft_a} = no\_failure \wedge p = (j_1 + j_2 \bmod 2)) \vee \\
&\qquad (\mathrm{ft_a} = stuck\_at\_zero \wedge p = 0)) \\
&\Rightarrow\; (\exists\, *, o \bullet (o = (i_1 + i_2 \bmod 2)) \wedge \\
&\qquad ((\mathsf{true} \wedge \\
&\qquad\quad ((\mathrm{ft_a} = no\_failure \wedge (i_1 + j_1 + i_2 + j_2 = 0 \bmod 2) \wedge o = p) \vee \\
&\qquad\quad (\mathrm{ft_a} = no\_failure \wedge (i_1 + j_1 + i_2 + j_2 = 1 \bmod 2) \wedge o = \neg p))) \\
&\qquad \vee \\
&\qquad (\mathrm{ft_a} = stuck\_at\_zero \wedge o = (i_1 + i_2 \bmod 2) \wedge p = 0)))
\end{aligned}
$$

Finally we can retrench the *Main* operation thus:

$$
\begin{array}{l}
G_{Main}(*,*) \;=\; \mathsf{true} \\
P_{Main}((r_1,r_2),(s_1,s_2),*,*) \;=\; (r_1 = s_1 \wedge r_2 = s_2) \\
O_{Main}(adder, sum; *,*,*,*,(r_1,r_2),(s_1,s_2)) \;= \\
\quad (\mathrm{ft}_1 = \mathrm{ft}_2 = \mathrm{ft}_\mathrm{a} = no\_failure \wedge adder = sum) \\
C_{Main}(*,*,\mathrm{adder}, sum; *,*,(r_1,r_2),(s_1,s_2)) \;= \\
\quad (\mathrm{ft}_\mathrm{a} = stuck\_at\_zero \wedge sum = 0) \vee \\
\quad (\mathrm{ft}_\mathrm{a} = no\_failure \wedge \mathrm{ft}_1 \neq \mathrm{ft}_2 \wedge \mathrm{adder} = \neg sum) \vee \\
\quad (\mathrm{ft}_\mathrm{a} = no\_failure \wedge \mathrm{ft}_1 = inverted = \mathrm{ft}_2 \wedge \mathrm{adder} = sum)
\end{array}
$$

Again, when we substitute these into the PO we get the messier but still easily verified:

$$
\begin{array}{l}
\mathsf{true} \wedge (r_1 = s_1 \wedge r_2 = s_2) \wedge \\
(\exists\, c_1, c_2 \bullet \\
\quad ((\mathrm{ft}_1 = no\_failure \wedge c_1 = s_1) \vee (\mathrm{ft}_1 = inverted \wedge c_1 = \neg s_1)) \wedge \\
\quad ((\mathrm{ft}_2 = no\_failure \wedge c_2 = s_2) \vee (\mathrm{ft}_2 = inverted \wedge c_2 = \neg s_2)) \wedge \\
\quad ((\mathrm{ft}_\mathrm{a} = no\_failure \wedge sum = (c_1 + c_2 \bmod 2)) \vee \\
\quad (\mathrm{ft}_\mathrm{a} = stuck\_at\_zero \wedge sum = 0))) \\
\Rightarrow\; (\exists\, *, \mathrm{adder} \bullet \\
\quad (\exists\, b_1, b_2 \bullet r_1 = b_1 \wedge r_2 = b_2 \wedge (\mathrm{adder} = (b_1 + b_2 \bmod 2))) \wedge \\
\quad ((\mathsf{true} \wedge \mathrm{ft}_1 = \mathrm{ft}_2 = \mathrm{ft}_\mathrm{a} = no\_failure \wedge \mathrm{adder} = sum) \\
\qquad\quad \vee \\
\quad ((\mathrm{ft}_\mathrm{a} = stuck\_at\_zero \wedge sum = 0) \vee \\
\quad (\mathrm{ft}_\mathrm{a} = no\_failure \wedge \mathrm{ft}_1 \neq \mathrm{ft}_2 \wedge \mathrm{adder} = \neg sum) \vee \\
\quad (\mathrm{ft}_\mathrm{a} = no\_failure \wedge \mathrm{ft}_1 = inverted = \mathrm{ft}_2 \wedge \mathrm{adder} = sum))))
\end{array}
$$

5 Fault Trees and Compositions

Associated with the retrenchment PO is the (one step) retrenchment simulation relationship, obtained from the PO by replacing the top level implication by a conjunction, and removing the existential quantification. It describes those pairs of steps of which the PO speaks, which make its antecedent valid:

$$
\begin{array}{l}
G(u,v) \wedge P_{Op}(i,j,u,v) \wedge Op_C(v,j,v',p) \wedge Op_A(u,i,u',o) \;\wedge \\
((G(u',v') \wedge O_{Op}(o,p;u',v',u,v,i,j)) \vee C_{Op}(u',v',o,p;u,v,i,j))
\end{array}
$$

and is written $(u \text{ -}(i, Op_\mathrm{A}, o)\text{->} u')\ \Sigma^1\ (v \text{ -}(j, Op_\mathrm{C}, p)\text{->} v')$. We will now show that in the present context it can be used to extract fault trees for the *Bit* operation.

In a typical fault of Bit_C the input is 0 while the output is 1, in contrast to the ideal behaviour of Bit_A which has 0 for both. We know already that these

abstract and concrete values validate the PO. Let Σ^1_{Bit} be the simulation relation for *Bit* (which is derived from the previous section's PO verification condition for *Bit* by applying the syntactic modifications mentioned). We conjoin expressions defining the values of i, j, o as 0 and p as 1, to Σ^1_{Bit}, and obtain:

$$i = 0 \land o = 0 \land j = 0 \land p = 1 \land \Sigma^1_{Bit}$$

Applying the substitutions and simplifying, we infer:

$$\mathsf{true} \land \mathsf{true} \land ((\mathrm{ft} = no\_failure \land \mathsf{false}) \lor (\mathrm{ft} = inverted \land \mathsf{true})) \land$$
$$\mathsf{true} \land ((\mathsf{true} \land \mathrm{ft} = no\_failure \land \mathsf{false}) \lor (\mathrm{ft} = inverted \land \mathsf{true}))$$

The only way that this can be true is if $\mathrm{ft} = inverted$ holds. We have derived the cause of the fault from the definition of the fault's behaviour. A fault tree could now be constructed with the fault definition as top level event and its inferred cause, $\mathrm{ft} = inverted$, as its child.

A similar technique could yield a fault tree for the *Adder* operation, consisting of a single cause $\mathrm{ft_a} = stuck\_at\_zero$, if the fault concerned concrete inputs $0, 1$ say and concrete output 0.

It will not have escaped the reader's notice that we are performing a kind of resolution to derive the fault tree. How then does it fare with compound operations such as *Main*? Operations such as *Main* raise the question of how to understand *Main*'s retrenchment in the context of the retrenchments of its components. Preferably, we want to understand the former in terms of a parallel and sequential composition of the latter. Now, sequential composition of retrenchments has been explored in some depth in [22] but it turns out that a completely different notion of sequential composition is appropriate here.

We require a notion of composition that gathers any of the erroneous or degraded cases that arise during the information flow, into the concession of the composed retrenchment, leaving only the completely fault-free cases for the output relation. That there is some choice in the matter arises because, in this paper, the information flow is via the inputs and outputs rather than the state. Since O and C are in disjunction, a true fact about I/O can be accomodated in either O or C without changing the value of $O \lor C$. Several things conspire to make our goal an achievable one:

- Each operation is a total relation.
- In each operation the various correct and degraded cases are disjoint.
- The various output and concedes relations have been carefully structured.
- *Adder*'s within relation has deliberately been made unrestrictive.

Suppose we sequentially compose two retrenchments, each with trivial state $*$. Dropping the true retrieve relation and all mention of the state for economy's sake, the first retrenchment will give rise to a simulation relationship:

$$P^{\mathrm{I}}_{Op}(i, j) \land Op^{\mathrm{I}}_{C}(j, c) \land Op^{\mathrm{I}}_{A}(i, b) \land (O^{\mathrm{I}}_{Op}(b, c; i, j) \lor C^{\mathrm{I}}_{Op}(b, c; i, j))$$

and the second one, a simulation relationship:

$$P^{\mathrm{II}}_{Op}(b, c) \land Op^{\mathrm{II}}_{C}(c, p) \land Op^{\mathrm{II}}_{A}(b, o) \land (O^{\mathrm{II}}_{Op}(o, p; b, c) \lor C^{\mathrm{II}}_{Op}(o, p; b, c))$$

The distributive law applied to the conjunction of these yields a simulation:

$$P^{\mathrm{I;II}}_{Op}(i,j) \wedge Op^{\mathrm{I;II}}_{C}(j,p) \wedge Op^{\mathrm{I;II}}_{A}(i,o) \wedge (O^{\mathrm{I;II}}_{Op}(o,p;i,j) \vee C^{\mathrm{I;II}}_{Op}(o,p;i,j))$$

where we define:

$$\begin{array}{ll} Op^{\mathrm{I;II}}_{A} = Op^{\mathrm{I}}_{A}; Op^{\mathrm{II}}_{A} & P^{\mathrm{I;II}}_{Op} = P^{\mathrm{I}}_{Op} \\ Op^{\mathrm{I;II}}_{C} = Op^{\mathrm{I}}_{C}; Op^{\mathrm{II}}_{C} & O^{\mathrm{I;II}}_{Op} = O^{\mathrm{I}}_{Op}; O^{\mathrm{II}}_{Op} \\ & C^{\mathrm{I;II}}_{Op} = O^{\mathrm{I}}_{Op}; C^{\mathrm{II}}_{Op} \vee C^{\mathrm{I}}_{Op}; O^{\mathrm{II}}_{Op} \vee C^{\mathrm{I}}_{Op}; C^{\mathrm{II}}_{Op} \end{array}$$

with ; denoting the usual composition of relations. Provided that $(O^{\mathrm{I}}_{Op} \vee C^{\mathrm{I}}_{Op}) \Rightarrow P^{\mathrm{II}}_{Op}$ then all composable individual steps described by the individual retrenchments will survive to the composition. This simulation relationship can be understood as arising from a composed retrenchment with data $P^{\mathrm{I;II}}_{Op}, O^{\mathrm{I;II}}_{Op}, C^{\mathrm{I;II}}_{Op}$ and trivial $G^{\mathrm{I;II}}$. We re-emphasise that this is not the only viable definition of sequential composition for retrenchments, particularly in view of the special conditions that have to hold for it to be well defined.

Parallel composition is similar and easier. Again we have a conjunction, this time of simulation relationships acting on disjoint spaces. Denoting parallel composition by | (logically a conjunction), we get a simulation relationship corresponding to the data:

$$\begin{array}{ll} Op^{\mathrm{I|II}}_{A} = Op^{\mathrm{I}}_{A} | Op^{\mathrm{II}}_{A} & P^{\mathrm{I|II}}_{Op} = P^{\mathrm{I}}_{Op} | P^{\mathrm{II}}_{Op} \\ Op^{\mathrm{I|II}}_{C} = Op^{\mathrm{I}}_{C} | Op^{\mathrm{II}}_{C} & O^{\mathrm{I|II}}_{Op} = O^{\mathrm{I}}_{Op} | O^{\mathrm{II}}_{Op} \\ & C^{\mathrm{I|II}}_{Op} = O^{\mathrm{I}}_{Op} | C^{\mathrm{II}}_{Op} \vee C^{\mathrm{I}}_{Op} | O^{\mathrm{II}}_{Op} \vee C^{\mathrm{I}}_{Op} | C^{\mathrm{II}}_{Op} \end{array}$$

It is not hard to see that doing the above for two *Bit*s composed in parallel, with the outcome sequentially composed with an *Adder*, yields retrenchment data equivalent (in the variables' and other symbols' natural interpretations[2]) to that given for *Main*[3].

We note that if each of $O^{\mathrm{I}}_{Op}, C^{\mathrm{I}}_{Op}, O^{\mathrm{II}}_{Op}, C^{\mathrm{II}}_{Op}$, is a disjunction of cases, then all of $C^{\mathrm{I;II}}_{Op}, C^{\mathrm{I|II}}_{Op}$ proliferate the case analysis via the distributive law. With this observation, we can outline the construction of multilevel fault trees from a composed retrenchment as follows, using our *Main* operation with abstract inputs $r_1 \neq r_2$ and concrete output $sum = 0$ as a running example.

Our technique is to resolve the values defining the fault with the composed simulation relation Σ^{1}_{Main}, to yield any intermediate values needed, and apply these to the decomposition of the composed concedes relation whose structure

[2] Note that we are *not* claiming propositional equivalence here.

[3] We emphasise once more that this is a consequence of *design*, and the design of the retrenchment data for *Main* in particular. It is very easy to write down different retrenchment data for *Main*, which are equally adept at discharging the retrenchment operation PO for *Main*, but which do not arise as the composition of the retrenchment data for *Main*'s subcomponents.

will reveal the required fault tree. We need to work with both Σ^1_{Main} and C_{Main} as the latter need not contain all the data required during the decomposition.

The top level constructor of our system is a sequential composition, so the top level event of the fault tree corresponds to a collection of values that makes the composed simulation relation $(* \text{-}(i, Op_{\mathrm{A}}^{\mathrm{I;II}}, o) \text{-}> *)\ \Sigma^1\ (* \text{-}(j, Op_{\mathrm{C}}^{\mathrm{I;II}}, p) \text{-}> *)$, and specifically the concession $C_{Op}^{\mathrm{I;II}}$, valid. Here II refers to *Adder* and I refers to two *Bit*s in parallel. If $C_{Op}^{\mathrm{I;II}}$ is a disjunction of cases, the ones that are `true` in the given valuation are the possible alternative causes of the fault, and lead to a disjunctive branching at the next level of the fault tree.

In our example we have two validated alternatives in C_{Main}, namely ($\mathrm{ft_a} = stuck\_at\_zero \wedge sum = 0$) and ($\mathrm{ft_a} = no\_failure \wedge \mathrm{ft}_1 \neq \mathrm{ft}_2 \wedge adder = \neg sum$). The first of these gives a bottom level explanation of the fault and needs no further analysis, closing off that branch of the fault tree. The second asserts $\mathrm{ft_a} = no\_failure$ for the *Adder* component, and so does need further analysis.

We now decompose the composed simulation relation in a reversal of the process described above. This involves finding intermediate abstract and concrete values b_1, b_2, c_1, c_2, that can act as existential witnesses for both the (de)composed simulation relation, and for the decomposition of $C_{Bits}^{\mathrm{I}}; O_{Adder}^{\mathrm{II}}$, where C_{Bits}^{I} reduces to $O_{Bit_1} | C_{Bit_2} \vee C_{Bit_1} | O_{Bit_2}$. This gives a three way disjunctive structure at the top level of the fault tree.[4] (The third disjunct of C_{Bits}^{I} namely $C_{Bit_1} | C_{Bit_2}$ is excluded by $\mathrm{ft}_1 \neq \mathrm{ft}_2$ which we just derived.)

Summarising, either the *Adder* failed, or one but not the other of the two *Bit*s failed. Aggregating the intermediate value cases which yield the same fault tree structure, each of the latter two options can now be seen as a conjunction of three facts:

1. $Bit_{(1+k)}$ failed.
2. $Bit_{(1+(1-k))}$ functioned correctly.
3. The *Adder* functioned correctly.

The first two conjuncts come from the parallel decomposition of C_{Bits}^{I} (which is a conjunction), while the third comes from the instantiation of the existential witness b_1, b_2, c_1, c_2, during the sequential decomposition of $C_{Bits}^{\mathrm{I}}; O_{Adder}^{\mathrm{II}}$ (this being $(\exists b_1, b_2, c_1, c_2 \bullet C_{Bits}^{\mathrm{I}}((b_1, b_2), (c_1, c_2), \ldots) \wedge O_{Adder}^{\mathrm{II}}((b_1, b_2), (c_1, c_2)))$.

With these three alternatives, we have derived the structure of the fault tree for this example as generated by the FSAP/NuSMV-SA toolkit [11]. See Fig. 1 (reproduced, with the authors' permission). It is not hard to see that the techniques described can be applied to more deeply nested compositions, to give derivations of fault trees for faults that occur deeper in the structure of a complex system. The derivation of the fault trees that we have outlined gives rise to fresh validation opportunities, by comparing this kind of derivation with more conventional routes.

[4] We have silently merged cases in which distinct data values lead to the same fault tree, for clarity of exposition.

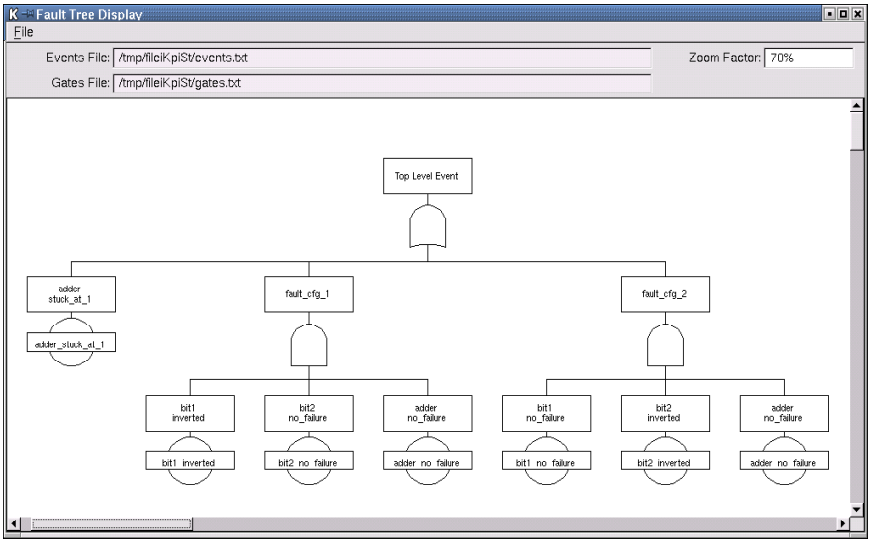

Fig. 1. The fault tree for the *Adder* example, from [11].

6 Model Checking

The fault tree analysis above was sensitive to the temporal order of the component transitions in a sequential composition, via the before/after ingredients of the retrenchment simulation relationship. A similar sensitivity is a feature of the model checking of temporal properties, so we can easily imagine that a retrenchment approach could also prove profitable in that sphere. Once more following [11], the ideal *Main* operation satisfies:

$$\mathbf{AG}(r_1 = 0 \wedge r_2 = 0 \ \rightarrow \ adder = 0)$$

while the degraded system satisfies:

$$\mathbf{AG}(s_1 = 0 \wedge s_2 = 0 \wedge sum \neq 0 \ \rightarrow \ (\mathrm{ft}_1 = inverted \vee \mathrm{ft}_2 = inverted))$$

and these are checkable via the model checker in the FSAP/NuSMV-SA safety analysis platform. Conventional formal techniques (i.e. refinement) cannot relate these two facts. However, in these two facts, it is not hard to recognise a very similar situation to the one analysed above, and so with retrenchment composition/decomposition techniques, we can expect a similar degree of success to that which we enjoyed for fault trees. To be sure there are technical issues to do with the integration of the retrenchment viewpoint and the temporal viewpoint, which will clutter the development somewhat, but these will not be onerous. The resulting analysis leads to fresh validation opportunities, as above.

7 Conclusions

In the preceding sections we have outlined a transition system model of fault injection similar to the one used in [11], and shown that while refinement struggles to describe the fault injection process, retrenchment accomplishes this in a natural manner. Of course the two faults we explored in detail are merely representative, and other typical faults dealt with in the FSAP/NuSMV-SA safety analysis platform, such as *random* or *glitch* could be accomodated without problems.

With fault injection under control, we indicated how retrenchment composition and decomposition could lead to the generation of fault trees for the simple *Adder* system. Retrenchment composition/decomposition is investigated in [20], [21], and the application of comparable ideas here is a gratifying endorsement of this family of techniques. Of course there is more left out than discussed, and a full treatment of the relationship between retrenchment and fault trees will be given elsewhere. (For instance the distinct results that can arise from monotonic versus non-monotonic analysis when faulty and non-faulty cases overlap can be brought out by finetuning the fault tree extraction algorithm.) Similar remarks apply to the promising interaction between retrenchment and model checking, which will also be pursued elsewhere.

Acknowledgements The authors would like to express their thanks to Marco Bozzano and Adolfo Villafiorita for their feedback on an earlier version of this paper.

References

1. J R Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. R Banach. Retrenchment and system properties. Submitted.
3. R Banach and C Jeske. Output retrenchments, defaults, stronger compositions, feature engineering. Submitted.
4. R Banach and M Poppleton. Engineering and theoretical underpinnings of retrenchment. Submitted.
5. 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.
6. R Banach and M Poppleton. Retrenchment and punctured simulation. *Proc. IFM-99, Springer*, Araki, Gallway, Taguchi (eds.):457–476, 1999.
7. R Banach and M Poppleton. Sharp retrenchment, modulated refinement and punctured simulation. *Form. Asp. Comp.*, 11:498–540, 1999.
8. R Banach and M Poppleton. Retrenching partial requirements into system definitions: A simple feature interaction case study. *Requirements Engineering Journal*, 8:266–288, 2003.
9. E Boiten and J Derrick. Io-refinement in Z. In A Evans, D Duke, and T Clark, editors, *Electronic Workshops in Computing.* Springer-Verlag, September 1998. Proc. Third BCS-FACS Northern Formal Methods Workshop. Ilkley, U.K.

10. J P Bowen and S Stavridou. Formal methods and software safety. In H. H. Frey, editor, *Safety of Computer Control Systems (SAFECOMP)*, pages 93–98. Pergamon Press, October 1992. Proc. IFAC Symposium, Zurich, Switzerland.
11. M Bozzano and A Villafiorita. Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. *Computer Safety, Reliability, and Security, LNCS*, 2788:49–62, 2003.
12. M Bozzano and A Villafiorita. Integrating fault tree analysis with event ordering information. *Proc. ESREL 2003*, pages 247–254, 2003.
13. 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.
14. M Bozzano, A Villafiorita, et al. 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.
15. W P de Roever and K Engelhardt. *Data Refinement Model-Oriented Proof methods and their Comparison.* Cambridge University Press, 1998.
16. J Derrick and E Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Springer-Verlag UK, 2001.
17. J Jacky. *The Way of Z.* Cambridge University Press, 1997.
18. S Liu and R Adams. Limitations of formal methods and an approach to improvement. *Proc. 1995 Asia-Pacific Software Engineering Conference (APSEC'95), IEEE Computer Society Press, Brisbane, Australia*, pages 498–507, December 1995.
19. S Liu, V Stavridou, and B Duterte. The practice of formal methods in safety-critical systems. *The Journal of Systems and Software*, 28(1):77–87, January 1995.
20. M Poppleton and R Banach. Structuring retrenchments in B by decomposition. *International Symposium of Formal Methods Europe (FME 2003), Pisa, Italy, LNCS*, 2805:814–833, September 2003.
21. M Poppleton and R Banach. Requirements validation by lifting retrenchments in B. In *Proc. 9th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-04), Florence, Italy.* IEEE Computer Society Press, 2004. to appear.
22. M R Poppleton. *Formal methods for Continuous Systems: Liberalising Refinement in B.* PhD thesis, University of Manchester, Computer Science Dept., 2001.
23. B Potter, J Sinclair, and D Till. *An Introduction to Formal Specification and Z.* Prentice Hall, second edition, 1996.
24. S Schneider. *The B-Method: An Introduction.* PALGRAVE, 2001.
25. E Sekerinski and K Sere. *Program Development by Refinement: Case Studies Using the B-Method.* Springer, 1998.
26. J M Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.
27. J Woodcock and J Davies. *Using Z, Specification, Refinement and Proof.* Prentice Hall, 1996.
28. J C P Woodcock and C C Morgan. Refinement of state-based concurrent systems. *Formal Methods in Software Development, LNCS*, 428, 1990.