Automatic Generation of Minimal Cut Sets

Sentot Kromodimoeljo and Peter A. Lindsay

School of IT&EE, The University of Queensland, St Lucia Qld 4072, Australia

A cut set is a collection of component failure modes that could lead to a system failure. Cut Set Analysis (CSA) is applied to critical systems to identify and rank system vulnerabilities at design time. Model checking tools have been used to automate the generation of minimal cut sets but are generally based on checking reachability of system failure states. This paper describes a new approach to CSA using a Linear Temporal Logic (LTL) model checker called BT Analyser that supports the generation of multiple counterexamples. The approach enables a broader class of system failures to be analysed, by generalising from failure state formulae to failure behaviours expressed in LTL. The traditional approach to CSA using model checking requires the model or system failure to be modified, usually by hand, to eliminate already-discovered cut sets, and the model checker to be rerun, at each step. By contrast, the new approach works incrementally and fully automatically, thereby removing the tedious and error-prone manual process and resulting in significantly reduced computation time. This in turn enables larger models to be checked. Two different strategies for using BT Analyser for CSA are presented. There is generally no single best strategy for model checking: their relative efficiency depends on the model and property being analysed. Comparative results are given for the A320 hydraulics case study in the Behavior Tree modelling language.

Keywords: Behavior Trees; minimal cut sets; model checking; safety analysis

1 Introduction

A *cut set* is a collection of component failures that could lead to a system failure. A cut set is *minimal* if none of its proper subsets are themselves cut sets. Cut Set Analysis (CSA) is the discovery of a complete set of minimal cut sets (MCSs) for given system failure modes. CSA, or an equivalent method such as Fault Tree Analysis (FTA), is typically mandated by standards for critical systems (e.g. [11]) to identify and rank system vulnerabilities at design time.

Traditionally, a system failure mode, or *top event* to give it its technical name, has to be identified before CSA can proceed. Model checkers are often used to automate CSA when the top event can be characterised by a state formula. CSA proceeds by determining if the top event is reachable for various component failure-mode combinations (the cut sets). Of particular interest are failure combinations that are minimal (the MCSs). For some modelling notations, CSA has been fully automated [2, 3, 4].

Rae and Lindsay [26] generalised the characterisation of system failure in FTA to violation of a temporal property rather than simply a state formula. This approach enables a broader class of system failures to be examined, such as state changes under circumstances when states shouldn't change, and action sequences occurring in an undesirable order. While such properties can sometimes be captured by adding an observer automaton to the model, it is often more natural to state the property as a temporal property, and desirable not to modify the model [16]. And in other examples the failures themselves are actually behaviors rather than events or conditions: for example Cerone *et al* [6] classify human failures by repeated patterns of behaviour, in a highly interleaved cognitive task where it is impossible to say exactly when the failure occurred.

Lindsay *et al* [17] developed a semi-automated approach to CSA with this richer notion of failures, using Behavior Trees (BT) to model the system with potential component failure-behaviours injected.

The SAL model checker [20] was used iteratively to identify MCSs, with the temporal property reformulated manually at each step to ignore previously discovered MCSs. The manual steps were tedious and error-prone. Moreover, in order to generate a complete CSA, the model structure needed to be "flat", in the sense that if failure event $C1$ is preceded by failure event $C2$ in some path, then there also exists a path in which $C1$ occurs but $C2$ does not. (This is a stronger assumption than simply that failures are independent.)

This paper describes a completely new approach to automation of CSA, using a new model checker called BT Analyser, in which search for counterexamples can be directed [13]. As well as avoiding the need for manual steps and the assumption about non-dependence of component failures, the checker is significantly more efficient than the one used in [17], with capabilities that go well beyond CSA. The automation takes advantage of some novel features of BT Analyser including the use of *cycle constraints* for counterexample generation, *global constraints*, and incremental analysis, resulting in an efficient tool for CSA. This paper also compares alternative strategies for the automatic generation of minimal cut sets to illustrate the virtues of the novel features of BT Analyser. The approach is illustrated on a BT model to enable comparison with previous approaches [17] but it is applicable to any modelling notation that can be translated to a finite state transition system.

- Section 2 describes the modelling framework and introduces CSA terminology and concepts.

- Section 3 describes the LTL model checker for Behavior Trees.

- Section 4 describes the techniques for generating minimal cut sets.
    - Section 4.1 describes how a cut set is extracted from a counterexample path.
    - Section 4.2 describes alternative approaches for verifying the minimality of a cut set.
    - Section 4.3 discusses two strategies for automating the generation of all minimal cut sets.

- Section 5 presents results of experiments on applying the techniques to the Airbus A320 hydraulic system case study used in [3] and [17].

- Section 6 discusses application of the approach to other modelling notations.

- Sections 7, 8 and 9 discusses related work, provides a summary and conclusion, and discusses future work, respectively.

2   Terminology and assumptions

The core of BT Analyser works for a very general class of modelling languages, including asynchronous finite-state systems with interleaving semantics [13]. It does so by using a modelling framework based on typed multi-variable state transition systems in which the effect of each transition is deterministic (but the choice of transition is non-deterministic) and the truth value of each atomic proposition in a state is determinable. Modelling languages that can be translated into this framework include state machines, Behavior Trees and activity diagrams.

In what follows we assume the component failure modes of interest have already been injected into the model of system behaviours: that is, the analyst has determined all of the component failure modes that are going to be analysed and has included events (called *basic events* in CSA and FTA) corresponding to occurrences of component failures. The effects of a component failure may need to be reflected in the system behaviour. However, we do not assume that a failed component remains failed. As usual for causal analysis techniques such as CSA and FTA, we assume component failures are independent: common mode failures are analysed after CSA [14].

For the purpose of CSA, we assume that basic events are encoded using special Boolean state variables, called *basic event flags* below. Each of these state variables represents the occurrence of the corresponding event. Initially, the state variable must have the value *false*. When the corresponding event occurs, the value of the state variable must change to *true* and no behaviour can change the value back to *false*. If the analyst's modelling notation does not support this, it can be easily added during translation to BT Analyser's modelling framework. Note that it is the event that cannot be undone, not the failure of a component.

Given a system safety property *Safe* expressed in Linear Temporal Logic (LTL) [25], the model checker investigates whether there is a *counterexample* to *Safe*: that is, a (possibly infinite) sequence π of transitions through the model which satisfies $\neg Safe$. (In fact the model checker looks for paths with a finite prefix p and an infinitely repeated subpath c, but if there is a counterexample at all then there is always one in the above form.) Note that *safety property* in this paper means the formalisation of a system safety requirement, rather than the narrow technical sense of $\mathbf{G}\neg P$ for a state condition P.

With the top event replaced by "a violation of system safety property", a cut set is the set of basic events that occur in a behaviour that violates the system safety property (a counterexample path for *Safe*). The set of basic events that occur in a counterexample path can be extracted from the cycle part of the counterexample path: they are exactly the basic events whose corresponding basic event flags have the value *true* in the cycle (recall that basic event flags can only change value from *false* to *true*, so in a cycle they must maintain their values).

Two different strategies are given in Section 4 below for generating counterexamples corresponding to MCSs.

3   BT Analyser: A Symbolic LTL Model Checker

BT Analyser is a symbolic LTL model checker for Behavior Trees. It implements novel techniques for LTL model checking and counterexample generation described in [13]. Although BT Analyser only accepts the Behavior Tree (BT) notation as the modelling notation, the core of the model checker is notation-independent.

Novel features of BT Analyser include:

- Directed counterexample path generation with cycle constraints and global constraints.

- Computation of fair states with global constraints.

- On-the-fly symbolic LTL model checking as well as the traditional fixpoint approach to symbolic model checking.

- Incremental analysis.

BT Analyser is a symbolic model checker, operating on sets of states characterised using propositions rather than individual states. A *symbolic state* is a proposition characterising a set of states.

BT Analyser's framework uses *elementary blocks* as abstract transitions. The effect of an elementary block transition is deterministic; the non-determinism is in choosing an elementary block to transition, when several are enabled.

For each elementary block b, let f_b be the function that computes the symbolic image under the transition (sub-)relation for b and let r_b compute the symbolic preimage. If B is the set of elementary blocks for the system modelled, then the function f_B for computing the symbolic image under the transition relation for the system is defined as

$$f_B(S) \triangleq \bigvee_{b\in B} f_b(S).$$

Similarly for the preimage,

$$r_B(S) \triangleq \bigvee_{b \in B} r_b(S).$$

Propositional operations are performed using ordered binary decision diagrams (OBDDs) [5].

Following model checking convention, let the negation of the system safety behaviour being analysed be denoted φ. During model checking, a *tableau* (decision graph) for φ is superimposed on the model producing an augmented model (see [13]). The image and preimage functions for the augmented model are denoted f_b' and r_b' for an elementary block b and f_B' and r_B' for the entire system.

Definition 1 (Symbolic Counterexample Path). A symbolic counterexample path has the form of a triple:

$$(I_\pi, p_\pi, s_\pi)$$

where I_π characterises a set of initial states, prefix p_π is a possibly empty finite sequence of elementary blocks, and cycle s_π is a non-empty finite sequences of elementary blocks that is repeated forever. A symbolic counterexample path represents a set of paths in the model that satisfy φ (the paths share the abstract transitions and all satisfy φ).

The intermediate symbolic states in a symbolic path can be computed from I_π and the image functions for the transitions (the elementary blocks) in the symbolic path.

Directed counterexample path generation allows BT Analyser to be told to find a symbolic counterexample path in which a cycle constraint cc is satisfied by a symbolic state in the cycle part of the path, and a global constraint gc is satisfied by all symbolic states in the path.

In the fixpoint approach to symbolic LTL model checking, the set of augmented states (states in the augmented model) that are fair with respect to φ is computed. A set of fairness constraints C_φ is defined in a manner that is dependent on the LTL encoding scheme. The fixpoint characterisation of the set of augmented states that are fair with respect to φ is

$$F_\varphi \triangleq \nu Z. \bigwedge_{c \in C_\varphi} r_B'(\mu Y.(Z \wedge c) \vee r_B'(Y))$$

where ν and μ are respectively the greatest fixpoint and least fixpoint operators of μ-calculus [12]. The inner least fixpoint expression characterises states that satisfy $Z \wedge c$ directly or do not satisfy $Z \wedge c$ but can reach states that satisfy $Z \wedge c$ in the augmented model, where Z is the variable of the outer greatest fixpoint operation. The outer greatest fixpoint operation interacts with the inner fixpoint operations, combining to ensure that each fairness constraint $c \in C_\varphi$ is satisfied infinitely often in each path whose states are in F_φ. (The overall fixpoint expression characterises states that can transition to states that can start paths in which each $c \in C_\varphi$ is satisfied infinitely often.)

A symbolic augmented state S_φ, defined according to the LTL encoding scheme, characterises the set of augmented states that are "commited" to starting paths that satisfy φ. Let I_φ denote the symbolic augmented state characterising the set of initial augmented states. The intersection characterised by

$$I_\varphi \wedge S_\varphi \wedge F_\varphi$$

determines if there is a counterexample for the LTL specification: the intersection is empty if and only if the LTL specification has no counterexamples.

BT Analyser allows the computation of augmented states that are fair with respect to φ while satisfying a global constraint gc globally. Note that in general this is not equivalent to $F_\varphi \wedge gc$. Instead, the following fixpoint characterisation can be used:

$$\nu Z. \quad gc \wedge \bigwedge_{c \in C_\varphi} r_B'(\mu Y.(Z \wedge c) \vee r_B'(Y)). \tag{1}$$

The intersection of the set of augmented states characterised by (1) with the set characterised by $I_\varphi \wedge S_\varphi$ determines if there is a counterexample path for the LTL specification for which *gc* holds globally.

In addition to the fixpoint approach to symbolic model checking, the model checker also allows on-the-fly symbolic LTL model checking. The algorithm used is an adaptation of the standard nested DFS algorithm [9] with the following important differences:

- The adapted algorithm works with symbolic transitions (using elementary blocks) and symbolic states.

- The adapted algorithm constructs the Büchi automaton implicitly and on-the-fly.

Details of the algorithm can be found in [13].

Finally, an important feature of BT Analyser is the ability to perform analysis incrementally. As an example, an analysis can start with reachability analysis, followed by computation of fair states, followed by computation of counterexample states (states that can be part of counterexample paths), followed by the generation of a counterexample path. We can then tell BT Analyser to find more counterexample paths as well as fair states with global constraints without redoing the initial parts of the analysis. This enables different strategies to be used for different problems. An example is provided in [13] where reachability analysis before model checking is a good strategy for a prioritised BT model (where internal system transitions are prioritised over external events) but is a bad strategy for a non-prioritised BT model.

4 Generating Minimal Cut Sets

We assume that component failure modes have been injected into the model. An LTL specification whose violation represents a system failure is first model checked. If a traditional top event represented by a state formula *top* is to be used, then the LTL specification model checked would be $\mathbf{G} \neg top$. Using our method, $\mathbf{G} \neg top$ can be replaced by any LTL formula representing non-occurrence of the behaviour associated with system failure.

In what follows let φ denote the hazardous system behaviour being analysed (i.e., the negation of the LTL formula representing the desired system safety property).

Cut sets will be extracted from counterexample paths and verified to be minimal. The entire process of generating all MCSs can be fully automated.

4.1 Extracting Cut Sets from Counterexample Paths

Traditionally, a cut set is the set of basic events that occur in a behaviour leading to the top event. Unfortunately, LTL does not include events. Even the more expressive temporal logic CTL* (see e.g., [8]) does not include events. The occurrence of an event is obviously irreversible: once an event occurs, from that point on, the event has occurred. A simple way to encode the occurrence of an event is to use a Boolean state variable, say *eOccurred*. Initially the value of *eOccurred* is *false*. When the event occurs, *eOccurred* is set to *true* and no behaviour can change it back to *false*. This way of encoding the occurrence of an event is used by Lindsay *et al* in their method [17]. We call the state variable for a basic event a *basic event flag*. A cut set can be encoded as a conjunction of the basic event flags that correspond to elements of the cut set.

We can extract the cut set by choosing a symbolic state in the cycle part and pick all positive occurrences of basic event flags. As an example, suppose there are 11 basic components that can fail and their

failure occurrences are represented by the basic event flags

$$distyF, distgF, distbF, E1F, E2F, PTUF, EDPyF, EDPgF, EMPbF, EMPyF, \text{ and } RATF.$$

We will denote the number of basic events *MAX*. Thus for the above example $MAX = 11$. Suppose further that we choose the following symbolic state from the cycle part of the symbolic counterexample path (taken from an actual run of the model checker on the A320 hydraulics case study):

$$
\begin{aligned}
& (Pilot = ready) \wedge (Engine1 = on) \wedge (Engine2 = off) \wedge (Yellow = off) \\
\wedge\ & (PTUy = off) \wedge (PTU = off) \wedge (EDPy = off) \wedge (EDPg = on) \wedge (Blue = off) \\
\wedge\ & (EMPy = off) \wedge (EMPb = off) \wedge (RAT = off) \wedge \neg E1F \wedge E2F \wedge \neg distyF \\
\wedge\ & \neg distgF \wedge \neg distbF \wedge PTUF \wedge \neg EDPyF \wedge \neg EDPgF \wedge EMPbF \wedge EMPyF \\
\wedge\ & \neg RATF \wedge (Green = on) \wedge (Aircraft = flyingSlow) \wedge (System = operating),
\end{aligned}
$$

then the cut set extracted from the counterexample path would be

$$\{E2F, PTUF, EMPbF, EMPyF\}.$$

Any symbolic state in the cycle can be chosen, thus we can always choose the starting symbolic state of the cycle (basic event flags can only transition from *false* to *true*, thus their values must remain constant throughout a cycle).

   If the symbolic state has a disjunction (i.e., the symbol "∨" occurs in the proposition that is the symbolic state), then the symbolic state can be normalised into an irredundant sum of product (ISOP) form and a cut set can be extracted from any of the disjuncts. In BT Analyser, ISOP forms are generated directly from OBDDs using the Minato-Morreale algorithm [18, 19].

4.2   Verifying Minimality of Cut Sets

In isolation, if a cut set is extracted from a counterexample path, then it can be checked for minimality by ensuring that there are no counterexample paths if any member of the cut set is "removed". This can be performed in the model checker by computing the set of states that are fair with respect to φ with a global constraint *gc* representing the removal of a member of the cut set. The computation is performed using (1). If the intersection of the resulting set of states with the set of initial states is empty, then the cut set is minimal. The idea goes as follows. Continuing with the example from Section 4.1, suppose the cut set of interest (denoted *cs*) is

$$cs = \{E2F, PTUF, EMPbF, EMPyF\}.$$

The *gc* is a conjunction of all the other components not failing together with at least one of the members of the cut set also not failing, and for the above *cs* we would have

$$
\begin{aligned}
gc \quad \equiv \quad & \neg distyF \wedge \neg distgF \wedge \neg distbF \wedge \neg E1F \wedge \neg EDPyF \wedge \neg EDPgF \wedge \neg RATF \\
& \wedge (\neg E2F \vee \neg PTUF \vee \neg EMPbF \vee \neg EMPyF).
\end{aligned}
$$

The conjunction $\neg distyF \wedge \neg distgF \wedge \neg distbF \wedge \neg E1F \wedge \neg EDPyF \wedge \neg EDPgF \wedge \neg RATF$ represents components that have not failed and $(\neg E2F \vee \neg PTUF \vee \neg EMPbF \vee \neg EMPyF)$ represents the removal of at least one component from *cs*. If the intersection of the result of (1) with the set of initial states is not empty, then there is a counterexample path with global constraint *gc*, meaning there is a proper subset of *cs* that is a cut set, and thus *cs* is non-minimal. Otherwise *cs* is minimal.

Computing the set of states that are fair with respect to φ with global constraint gc is usually much faster than computing the set of states that are fair with respect to φ without global constraints. However, often this is still more expensive than counterexample path generation. With some strategies to be discussed in Section 4.3, minimality need not be checked and the number of computations of fair states can be reduced.

4.3 Strategies for Generating Minimal Cut Sets

Two different strategies for automating CSA are given here: a naive strategy and a systematic strategy. The naive strategy finds MCSs in no specific order, while the systematic strategy finds MCSs in a non-decreasing order in terms of size.

4.3.1 Naive Strategy

The steps of this strategy are as follows:

1. Compute the set of fair states.

2. Set $gc \leftarrow true$ (i.e., no global constraints).

3. Find a counterexample path with global constraint gc and no cycle constraints.

4. If a counterexample path is found, extract a cut set and verify its minimality as described in Section 4.2. If it is not minimal then find a counterexample path for the minimality and extract a smaller cut set. This can be repeated until a minimal cut set is extracted. Add the MCS to the set of MCSs sets found and modify gc to rule out the found MCS from further consideration: add the negation of a representation of the MCS as a conjunct to gc. For the above example cut set, the negation is

$$\neg(E2F \wedge PTUF \wedge EMPbF \wedge EMPyF).$$

   Repeat from step 3.

5. Otherwise no counterexample path was found that satisfies gc globally and all minimal cut sets have been found.

A proof sketch of the correctness of the strategy is as follows:

- Each conjunct in gc is associated with an MCS already found and rules out any more cut sets that are subsumed by the MCS.

- In step 4, each time an MCS is found, an appropriate conjunct is added to gc.

- Nowhere else is a conjunct added to gc.

- If there are no counterexamples that satisfy gc then there are no cut sets except those that are subsumed by the MCSs found in step 4 (since gc rules out cut sets that are subsumed by the MCSs already found).

Since after step 5 we can conclude that all cut sets are subsumed by the MCSs found, the MCSs found are exactly all the MCSs.

4.3.2 Systematic Strategy

As mentioned in Section 4.2, verifying the minimality of a cut set by computing fair states with global constraints can be expensive. A more systematic strategy that could be more efficient would be to find all MCSs of size 0, then all MCSs of size 1, then all MCSs of size 2, and so on until we reach size *MAX*. This strategy uses the cycle constraint feature of directed counterexample generation:

1. Compute the set of fair states.

2. Set $gc \leftarrow true$ and $n \leftarrow 0$.

3. Find a counterexample path with global constraint gc and cycle constraint "there are n or less basic events". For example, suppose there are 3 basic events represented by *C1F*, *C2F* and *C3F*. For $n = 1$ the cycle constraint would be

$$\neg C1F \wedge \neg C2F \vee \neg C1F \wedge \neg C3F \vee \neg C2F \wedge \neg C3F.$$

   It may be easier to view the constraint as "at least $MAX - n$ basic events have not happened". It is equivalent to the standard "choosing $MAX - n$ out of MAX" combinatorial problem.

4. If a counterexample path is found, the extracted cut set would be a minimal cut set. Add the MCS to the set of MCSs found and modify gc to rule out the found MCS from further consideration (as in step 4 of the naive strategy). Repeat from step 3.

5. Otherwise no counterexample path was found that satisfies the constraint. If $n < MAX$, set $n \leftarrow n + 1$ and repeat from step 3. Otherwise $n \geq MAX$, and all MCSs have been found.

A proof sketch of the correctness of the strategy is as follows (it is slightly different than the one for the naive strategy):

- Each conjunct in gc is associated with an MCS already found and rules out any more cut sets that are subsumed by the MCS.

- Each time an MCS is found in step 3, an appropriate conjunct to rule out the MCS found from further consideration is added to gc in step 4.

- Nowhere else is a conjunct added to gc.

- If no more counterexample is found in step 3, then there are no cut sets of size $\leq n$ except those subsumed by the MCSs already found.

- A cut set found in step 3 is a minimal cut set: the case where $n = 0$ is trivial; for $1 \leq n \leq MAX$, we can assume that all cut sets of size $\leq n - 1$ are subsumed by the MCSs already found therefore any cut set found in step 3 is a minimal cut set (since they are not subsumed by the MCSs already found, thus they are not subsumed by any cut set of size $< n$).

- If there are no counterexamples that satisfy gc when $n = MAX$ then there are no cut sets of size $\leq MAX$ except those that are subsumed by the MCSs already found.

Since the size of a cut set cannot be greater than *MAX*, after step 5 with $n = MAX$ we can conclude that all cut sets are subsumed by the MCSs found, thus the MCSs found are exactly all the MCSs.

5 Experimental Results

Experiments were conducted to compare the performances of the two strategies described in Section 4.3. The naive strategy was then slightly modified, with on-the-fly symbolic LTL model checking replacing directed counterexample path generation, and the performance of the modified strategy compared with that of the original strategy.

Table 1: A320 Hydraulics Timing Results

Strategy	Initial MC run	MCS Generation	Total
Naive	8 minutes	57 minutes	65 minutes
Systematic	8 minutes	5 minutes	13 minutes

5.1 Comparison of Two Strategies

The two strategies described in Section 4.3 were applied to the A320 hydraulics case study used in [3] and [17]. The BT model with the injected faults is included as an example for BT Analyser, which can be accessed using the DOI 10.14264/uql.2015.16 (it is part of the supplementary material for [13]). The experiments were performed on a notebook computer with a 2.7GHz i7 CPU and 16GB of RAM running Ubuntu. The BT model is similar to the one used in [17]. The LTL specification used is

$$\mathbf{G}\left(\neg INIT \vee \mathbf{G}\left(\begin{array}{cl} & (Yellow = on) \wedge (Green = on) \\ \vee & (Yellow = on) \wedge (Blue = on) \\ \vee & (Green = on) \wedge (Blue = on) \end{array}\right)\right)$$

where $\mathbf{G}$ is the *globally* temporal operator ($\square$ or box), and

$$INIT \triangleq (Yellow = on) \wedge (Green = on) \wedge (Blue = on).$$

Both strategies produced the same set of MCSs, although the order in which the MCSs were found were different. Five 2-element MCSs were found:

$\{distyF, distbF\}, \{distyF, distgF\}, \{distgF, distbF\}, \{distyF, EMPbF\}$ and $\{distgF, EMPbF\}$.

There were ten 3-element MCSs found:

$\{distyF, PTUF, EDPgF\}, \{distbF, PTUF, EDPgF\}, \{distyF, E1F, PTUF\},$
$\{distbF, E1F, PTUF\}, \{PTUF, EDPgF, EMPbF\}, \{E1F, PTUF, EMPbF\},$
$\{EDPyF, EDPgF, EMPyF\}, \{E1F, E2F, EMPyF\}, \{E2F, EDPgF, EMPyF\}$
and $\{E1F, EDPyF, EMPyF\}$.

There were six 4-element MCSs found:

$\{distbF, PTUF, EDPyF, EMPyF\}, \{distgF, PTUF, EDPyF, EMPyF\},$
$\{distbF, E2F, PTUF, EMPyF\}, \{distgF, E2F, PTUF, EMPyF\},$
$\{PTUF, EDPyF, EMPbF, EMPyF\}$ and $\{E2F, PTUF, EMPbF, EMPyF\}$.

Table 1 shows the overall timing results for the two strategies. Both strategies have the same initial model checking time for the LTL specification which is 8 minutes. The subsequent MCS generation times differed substantially.

The use of cycle constraints drastically reduced the counterexample generation time. As Fig. 1 shows, the counterexample generation times for the naive strategy started at a high of 490 seconds and dropped significantly as more global constraints were added. In contrast, the counterexample generation times for the systematic strategy (which uses cycle constraints) did not vary as much and were quick at between 3 and 9 seconds (see Fig. 2).

The times for computing fair states with global constraints to check minimality did not vary so much, as can be seen in Fig. 3. They were between 40 and 60 seconds. In fact, the four cases where the times
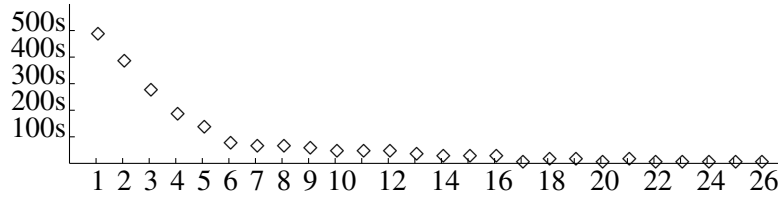
Figure 1: CTR Generation Time for Naive Strategy (26 counterexample paths)
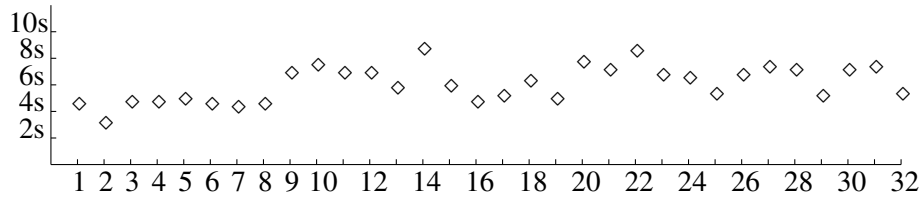

Figure 2: CTR Generation Time for Systematic Strategy (32 counterexample paths)

were significantly higher were exactly for cases where the cut set turned out to be non-minimal: cases 16, 19, 21 and 24. Interestingly, the performance penalty for those cases were somewhat offset by the significantly faster counterexample generation times that immediately followed: cases 17, 20, 22 and 25 in Fig. 1. There is no case 26 for checking minimality since no counterexample was found for the case.
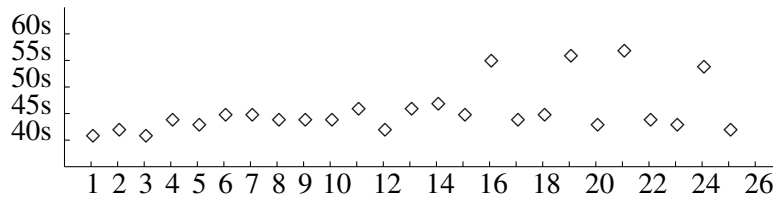

Figure 3: Minimality Checking Time for Naive Strategy (25 iterations)

The results from the experiments clearly show that cycle constraints in directed counterexample generation can drastically reduce the computation time required to find counterexample paths. Moreover, cycle constraints can be used to direct the counterexample path generation towards specific kinds of paths. For generating MCSs using the systematic strategy, the cycle constraint directs the model checker to find a counterexample path that produces a cut set of exactly the desired size.

The results also show the utility of global constraints. Global constraints serve two purposes:

- to rule out certain classes of counterexamples, and

- to reduce the search space for counterexamples.

The second is a pleasant consequence of the first. Generally, a smaller search space results in faster search time. The effect is rather dramatic in counterexample generation for the naive strategy (Fig. 1).

Finally, the results show the benefits of incremental analysis described in Section 3. Even a bad strategy (the naive strategy) is helped by incremental analysis. The computation time using the bad strategy is still better than the total computation time, using SAL (rerun on the same notebook computer used for the experiments), of the original approach in [17] (1 hour vs 4 hours). Although there are too many factors that are not taken into account for a fair comparison, it is clear that not redoing work can save a lot of time.

Table 2: Results for Naive Strategy with Symbolic On-the-fly LTL Checking

$|ICS|$	$|MCS|$	CTR Existence Test	Total OTF	Total Minimality Test
4	3	415.5s	6.8s	141s
6	3	414.0s	21.4s	246.7s
9	2	321.1s	72.2s	833.0s
8	2	271.8s	39.8s	738.1s
9	3	239.8s	81.8s	812.8s
8	2	346.7s	38.6s	781.6s
5	3	262.5s	23.3s	194.1s
5	2	254.3s	25.5s	361.7s
5	3	244.3s	23.5s	266.0s
7	2	230.6s	33.4s	615.0s
5	3	206.3s	24.6s	272.6s
5	3	210.2s	27.5s	196.9s
5	3	201.1s	27.9s	268.8s
6	3	191.6s	28.9s	385.9s
5	4	192.8s	22.2s	166.4s
6	4	190.0s	30.4s	279.2s
6	4	193.1s	28.2s	282.1s
4	3	191.2s	28.9s	128.6s
4	4	169.3s	26.4s	57.8s
5	4	164.7s	31.3s	167.5s
5	4	164.6s	33.4s	164.3s
		98.1s		
		5173.6s	676.0s	7360.1s

5.2 Using On-the-fly Symbolic LTL Model Checking

Recall from Section 3 that on-the-fly symbolic LTL model checking can be used to find counterexample paths quickly. However, the on-the-fly symbolic LTL model checking facility in BT Analyser is not as well developed as directed counterexample path generation. In particular the facility does not yet support cycle constraints. As a result, the facility cannot be used for the systematic strategy.

An experiment was conducted with on-the-fly symbolic LTL model checking replacing directed counterexample path generation for the naive strategy. In the experiment, the existence of any remaining counterexample path (and thus any remaining MCSs) was checked first in each round of MCS generation. This is because on-the-fly symbolic LTL checking might take a long time if there are no counterexamples.

Table 2 shows the results of the experiment with on-the-fly symbolic LTL checking. For each round (represented by a row), the column $|ICS|$ is for the size of the initial cut set found, the column $|MCS|$ is for the size of the MCS found, the column "CTR Existence Test" is for the time it took to check the existence of a counterexample path (by computing fair states), the column "Total OTF" is for the total of the on-the-fly symbolic LTL checking times, and the column "Total Minimality Test" is for the total of the minimality test times. The number of iterations for a round is $|ICS| - |MCS| + 1$, thus for $|ICS| = 9$ and $|MCS| = 2$, there would have been 8 iterations of on-the-fly symbolic LTL checking and minimality test in the round. The second last row represents the time needed to verify that there are no more MCSs.

The results show that, although the total of on-the-fly symbolic LTL checking times (11 minutes) is much less than the total of the counterexample generation times for the naive strategy with directed counterexample path generation (57 minutes), the overall analysis time is significantly greater. Even if we ignore the existence test times for the rounds and use 98.1 seconds as the time to check the absence of more counterexamples (we can imagine a parallelised implementation where on-the-fly symbolic LTL checking is run in parallel with checking the absence of counterexamples and whichever result is obtained first is used, with the other computation aborted), the total time is still around 135 minutes, compared to 65 minutes using directed counterexample path generation.

The disappointing performance when using on-the-fly symbolic LTL checking for the naive strategy appears to be caused by the *depth-first-search with imprecise (but safe) tracking of fairness constraints* nature of the algorithm used. This resulted in counterexample paths that tended to produce cut sets with many more failed components than necessary. Of the 21 MCSs, only one of them was found in one iteration, with the other 20 MCSs requiring 78 iterations in all. (By contrast, with directed counterexample path generation, 17 of the MCSs were found in one iteration and the remaining 4 were found in two iterations.) Thus, any savings from the fast counterexample generation times were completely wiped out by the times required for the minimality tests.

Perhaps if cycle constraints can be incorporated into on-the-fly symbolic LTL checking, then the systematic strategy might benefit from fast counterexample path generation times of on-the-fly symbolic LTL checking without the initial overhead of computing fair states.

6   Discussion

The approach described above, and BT Analyser, were developed in a very general typed multi-variable transition system modelling framework. The results were illustrated on a translation from the Behavior Tree notation into the framework but could easily be extended to support other modelling notations, including state-based notations such as VDM, Z and B. The main restriction is that all sets must be bounded finite sets and the number of threads is bounded. Note that if the model has a terminating state, then the state must be replaced by one that has a "null" transition (i.e., it can transition to itself). This ensures that all paths are infinite.

Fault injection in BT models is usually very straightforward. Although they are all in some sense equivalent, the BT model of a system can have different tree shapes depending on the order in which the subtrees corresponding to different requirements were integrated (see [10] for a description of the BT model development process). If they have been developed so that individual component behaviours are grouped together by component, which is very often the case, then fault injection is simply a matter of grafting a behaviour tree corresponding to the component failure mode into the tree, with the failure (basic) event at the graft point.

Failure Modes and Effects Analysis (FMEA) is a commonly used consequence analysis technique for critical systems. It consists of checking whether individual component failures can have hazardous effects on the system. Tool support can be provided for FMEA by injecting individual faults into system models and using model checking to see if undesirable system states are reachable. Grunske *et al* [15] reports experience automating FMEA using BT models and the SAL model checker. Although in principle this approach could be extended to CSA simply by injecting multiple faults into the model, one for each element of a potential cut set, in practice the limits of model checking are quickly reached due to the state explosion problem. By contrast, BT Analyser can handle system models like those in [15] in which all of the faults are injected, and it does so very efficiently.

The generalisation of *top event* from a failure state to a behaviour allows the approach to be applied to a broad class of modelling notations, including notations that do not explicitly represent state changes, such as the BT notation.

7   Related Work

Bieber *et al* [3] combine CSA and model checking in safety analysis of the A320 hydraulics case study. They use two different models, one of the static, physical architecture and an LTL model of the activation and failure behaviour of the system, and need reasoning to combine the results to prove the safety requirements, whereas our approach uses a single model and the entire analysis is automated.

Akerlund *et al* [2] propose that reliability analysis such as FTA and verification of safety requirements be performed on the same model. Their generation of MCSs uses the reachability analysis capability of a model checker, but is based on a state condition as the top event. The FSAP/NuSMV-SA safety analysis platform [4] and the DCCA method [21] also use the reachability analysis capabilities of model checkers to generate MCSs based on a state condition. In contrast, our approach is applicable when the top event is a behaviour.

Abdulla *et al* [1] use a SAT-based model checker to find MCSs for Scade models, in which component failure modes are modelled by replacing nominal flows by extended flows. Again, as with the approaches mentioned in the previous paragraph, they perform reachability analysis to a state condition.

Papadopoulos and Maruhn [23] construct a fault tree from the design model. Ortmeier and Schellhorn [22] model the fault trees themselves and verify their correctness and completeness against Statechart models. Cha *et al* [7] use the model checker UPPAAL to verify the correctness of fault trees against timed-automata models. In contrast, our approach does not need to involve fault trees.

Lindsay *et al* [17] use a model checker to generate MCSs from failure behaviour. However, some of the steps still need to be performed manually.

8   Summary and Conclusions

We have presented an approach that can be used to generate minimal cut sets automatically from a fault-injected model and a general safety requirement. The approach takes advantage of novel features of a new model checker, BT Analyser. In particular, the approach uses the incremental analysis capability of BT Analyser, directed counterexample path generation, cycle constraints and global constraints. The use of these features significantly reduces the computation time required when compared to the traditional way of using model checkers.

Experiments on an existing case study in the Behavior Tree notation show the effectiveness of the approach and the novel features of BT Analyser. The effectiveness of cycle constraints in producing counterexample paths with desired properties and in producing them quickly was a pleasant surprise. The use of constraints appears to help mitigate the state explosion problem in model checking.

9   Future Work

The on-the-fly symbolic LTL model checking facility in BT Analyser is not as developed as directed counterexample path generation. In particular, cycle constraints cannot be specified for counterexample

paths. A possible future work would be to incorporate cycle constraints into the on-the-fly symbolic LTL checking mechanism.

Another possible future work is to incorporate partial order reduction techniques [24] into on-the-fly symbolic LTL model checking in BT Analyser. This may speed-up on-the-fly checking in cases where there are no counterexamples.

10   Acknowledgements

References

[1] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren & O. Åkerlund (2006): *Designing Safe, Reliable Systems Using SCADE*. In: *Leveraging Applications of Formal Methods*, LNCS 4313, Springer, pp. 115–129, doi:10.1007/11925040_8.

[2] O. Akerlund, S. Nadjm-Tehrani & G. Stålmarck (1999): *Integration of Formal Methods into System Safety and Reliability Analysis*. In: *Proceedings of 17th International Systems Safety Conference*, pp. 326–336.

[3] P. Bieber, C. Castel & C. Seguin (2002): *Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System*. In: *Dependable Computing EDCC-4*, Springer, pp. 19–31, doi:10.1007/3-540-36080-8_3.

[4] M. Bozzano & A. Villafiorita (2007): *The FSAP/NuSMV-SA Safety Analysis Platform*. International Journal on Software Tools for Technology Transfer 9(1), pp. 5–24, doi:10.1007/s10009-006-0001-2.

[5] R.E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers C-35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[6] A. Cerone, S. Connelly & P. A. Lindsay (2008): *Formal analysis of human operator behavioural patterns in interactive surveillance systems*. Software and Systems Modeling 7(3), pp. 273–286, doi:10.1007/s10270-007-0072-x.

[7] S. Cha, H. Son, J. Yoo, E. Jee & P.H. Seong (2003): *Systematic Evaluation of Fault Trees using Real-time Model Checker UPPAAL*. Reliability Engineering & System Safety 82(1), pp. 11 – 20, doi:10.1016/S0951-8320(03)00059-0.

[8] E.M. Clarke, Jr., O. Grumberg & D.A. Peled (1999): *Model Checking*. MIT Press.

[9] C. Courcoubetis, M.Y. Vardi, P. Wolper & M. Yannakakis (1992): *Memory-Efficient Algorithms for the Verification of Temporal Properties*. Formal Methods in System Design 1(2/3), pp. 275–288, doi:10.1007/BF00121128.

[10] R.G. Dromey (2003): *From Requirements to Design: Formalizing the Key Steps*. In: *1st International Conference on Software Engineering and Formal Methods*, IEEE Computer Society, pp. 2–11, doi:10.1109/SEFM.2003.1236202.

[11] International Electrotechnical Commission (2010): *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Part 1: General requirements*. International Standard IEC 61508-1.

[12] D. Kozen (1983): *Results on the Propositional mu-Calculus*. Theoretical Computer Science 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[13] S. Kromodimoeljo (2014): *Controlling the Generation of Multiple Counterexamples in LTL Model Checking*. phdthesis, doi:10.14264/uql.2015.16.

[14] N. Leveson (1995): *Safeware - System Safety and Computers: A Guide to Preventing Accidents and Losses caused by Technology*. Addison-Wesley.

[15] L.Grunske, K. Winter, N. Yatapanage, S. Zafar,Saad & P.A. Lindsay (2011): *Experience with Fault Injection Experiments for FMEA*. Software: Practice and Experience 41(11), pp. 1233–1258, doi:10.1002/spe.1039.

[16] P.A. Lindsay, K. Winter & S. Kromodimoeljo (2012): *Model-based Safety Risk Assessment using Behavior Trees*. In: *Proceedings of the 6th Asia Pacific Conference on System Engineering*, Systems Engineering Society of Australia. Available at `http://staff.itee.uq.edu.au/pal/papers/SETE2012.pdf`.

[17] P.A. Lindsay, N. Yatapanage & K. Winter (2012): *Cut Set Analysis using Behavior Trees and Model Checking*. Formal Aspects of Computing 24(2), pp. 249–266, doi:10.1007/s00165-011-0181-8.

[18] S. Minato (1993): *Fast Generation of Prime-Irredundant Covers from Binary Decision Diagrams*. IEICE Transactions on Fundamentals of E76-A(6), pp. 967–973.

[19] E. Morreale (1970): *Recursive Operators for Prime Implicant and Irredundant Normal Form Determination*. IEEE Transactions on Computers 19(6), pp. 504–509, doi:10.1109/T-C.1970.222967.

[20] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea & A. Tiwari (2004): *SAL 2*. In: *16th International Conference on Computer Aided Verification*, LNCS 3114, Springer, pp. 496–500, doi:10.1007/978-3-540-27813-9_45.

[21] F. Ortmeier, W. Reif & G. Schellhorn (2006): *Deductive Cause-Consequence Analysis (DCCA)*. Proceedings of IFAC World Congress.

[22] F. Ortmeier & G. Schellhorn (2007): *Formal Fault Tree Analysis - Practical Experiences*. Electronic Notes in Theoretical Computer Science 185, pp. 139 – 151, doi:10.1016/j.entcs.2007.05.034.

[23] Y. Papadopoulos & M. Maruhn (2001): *Model-Based Synthesis of Fault Trees from Matlab-Simulink Models*. In: *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2001)*, IEEE Computer Society, pp. 77–82, doi:10.1109/DSN.2001.941393.

[24] D. Peled, T. Wilke & P. Wolper (1996): *An Algorithmic Approach for Checking Closure Properties of ω-Regular Languages*. In: *7th International Conference on Concurrency Theory*, LNCS 1119, Springer, pp. 596–610, doi:10.1016/S0304-3975(97)00219-3.

[25] A. Pnueli (1977): *The Temporal Logic of Programs*. In: *18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp. 46–57, doi:10.1109/SFCS.1977.32.

[26] A. Rae & P. Lindsay (2004): *A Behaviour-based Method for Fault Tree Generation*. In: *Proceedings of 22nd Internatonal System Safety Conference*, System Safety Society, pp. 289–298.