

## Safety assessment using Behavior Trees and Model Checking

Peter A. Lindsay    Kirsten Winter  
*School of IT&EE,  
 The University of Queensland,  
 Brisbane, Qld 4072, Australia  
 email: (p.lindsay,k.winter)@uq.edu.au*

Nisansala Yatapanage  
*Institute for Integrated and Intelligent Systems,  
 Griffith University,  
 Nathan, QLD 4111, Australia  
 email: n.yatapanage@griffith.edu.au*

**Abstract**—This paper demonstrates the use of Behavior Trees and model checking to assess system safety requirements for a system containing substantial redundancy. The case study concerns the hydraulics systems for the Airbus A320 aircraft, which are critical for aircraft control. The system design is supposed to be able to handle up to 3 different components failing individually, without loss of all hydraulic power. Verifying the logic of such designs is difficult for humans because of the sheer amount of detail and number of different cases that need to be considered. The paper demonstrates how model checking can yield insights into what combinations of component failures can lead to system failure.

**Keywords**—Formal modelling, automated failure analysis, model checking, safety requirements, cutset analysis, Behavior Trees

### I. INTRODUCTION

Systems with stringent fault-tolerance requirements, such as the safety critical systems that control aircraft flight, must be designed with high degrees of redundancy, so that if components fail then other components can take over and deliver the required functionality. Requirements of the form “at least  $N$  independent component failures are necessary before  $X$  system functions fail” are typical of the kinds of qualitative safety requirements that apply to critical systems in aerospace, automotive and other industries [1], [2], [3], [4]. Software is increasingly being used to control the configuration of such systems (e.g., in Modular Avionics [5]) and to undertake reconfiguration where necessary after components fail.

The requirement for redundancy management can add considerable complexity to system design. For example, more than 80% of the software used in modern flight management systems is typically concerned with redundancy management and fault tolerance. It is a complex and critical task to check correctness of (re)configuration logic by hand and to ensure that sufficient redundancy exists under all foreseeable circumstances and modes of operation. This seems like an ideal area for formal methods.

In this paper we demonstrate the use of model checking in conjunction with Behavior Trees [6], [7] to verify that a system design is sufficiently fault tolerant. We demonstrate a methodology for systematically using counterexamples to identify the combinations of component failures that could

lead to system failure. We argue that Behavior Trees simplify these tasks because they use a single model of system behaviour, sufficiently rich to capture the wide variety of component behaviours needed and investigate the effect of component failures on system behaviour, yet simple enough to state safety requirements naturally and transparently.

The approach is demonstrated on a case study from aircraft design: namely, the hydraulics systems of the Airbus A320 family of aircraft [8]. We chose this case study because it is reasonably manageable in size, yet complex and critical enough to demand very careful analysis. The hydraulics systems provide power to the devices which control key aircraft components in flight and on the ground (flaps, rudders, landing gear, etc) so it is absolutely critical that they work. The system design has considerable redundancy built into it, to try to ensure hydraulics is available despite subsystem failures (such as failure of the electricity supply, the distribution lines, the engines, etc.). In fact, this design is supposed to be able to handle up to 3 different individual components failing, without losing all hydraulics. The general problem is typical of the kinds of problems encountered in software-intensive critical systems.

The *Behavior Tree* (BT) notation [6], [7] is a graphical notation for modelling the functional requirements of a system provided in natural language. The strength of the BT notation is two-fold: Firstly, the graphical nature of the notation provides stakeholders with an intuitive understanding of the system as specified – an important factor especially for use in industry [9]. Secondly, the process of capturing requirements is performed in a stepwise fashion which provides a scalable solution for handling very large requirements specifications [6], [10]. The syntax and semantics of BTs have been formalised [11], [12] and an interface to the SAL tool suite [13] provides a means for their automated analysis [14].

The contribution of the current paper is to illustrate the ease with which system safety requirements can be formulated in this approach, and to demonstrate a systematic method for using counterexamples in order to generate *cutsets* for system models: that is, a complete analysis of the circumstances under which combinations of component failures could lead to hazardous system failures [15], [16].

We also illustrate the impact that alternative formulations can have on computation time. An associated paper compares the BT approach with other widely used system design methods [17].

The paper is structured as follows: Section II introduces the BT notation and model checking of BTs. Section III introduces the case study and explains the safety requirements. Section IV-A explains the BT model used here, along with assumptions, simplifications and differences from Bieber [8]. Section V explains the formalization of the safety requirements and gives details of properties used to validate the model. Section VI describes the use of counterexamples to discover the circumstances under which system failures may occur, and illustrates a process for determining cutsets and coeffectors. Section VII discusses the impact of relaxing some of the assumptions made in Section IV-A and Section VIII summarizes the paper and draws conclusions.

### A. Related Work

Model checking has been widely used in support of consequence analysis techniques such as Failure Modes and Effects Analysis (FMEA), in which faults are injected into a system design model and a model checker is used to see if they might lead to hazardous situations [14], [18], [19], [20], [21], [22]. It has also been used, with more limited success, in support of causal analysis techniques such as minimal cutset analysis and Fault Tree Analysis (FTA), in which the goal is to identify all possible circumstances under which hazardous situations might arise. (In the case of FTA the aim is also to reveal the “logic” of failure mechanisms, by tracing system failures back through design to find their root causes.) Some FTA approaches focus on automating construction of the fault tree from the design model [23] while others focus on formal modelling of fault trees and verification of their correctness and completeness [24].

Rauzy [25] advocates an approach to modelling the system using a formalism he calls mode automata, and then deriving cutsets from them as Boolean formulae. This is the approach underlying AltaRica [8], which has been supplemented with use of Aralia [26] to compute the prime implicants of hazards formulated as non-temporal failure conditions. Hammarberg *et al* demonstrate the use of Esterel to formally verify fault tolerance of a Field Programmable Gate Array (FPGA) for an aerospace application, by cutset extraction [27].

Generation and analysis of counterexamples from model checkers has a long and illustrious history [28]. In [14] we reported on manual inspection of counterexamples to generate scenarios for how a hazard might arise. Here we illustrate an extension of the BT notation with set-theoretic constructs which simplifies the representation of safety requirements, and describe a systematic approach to generation of minimal cutsets.

## II. PRELIMINARIES

### A. Behavior Trees

The syntax of the BT notation comprises nodes and edges. A node can be either a state realisation describing a state change of a component or one of its attributes, or a selection, guard, or event, guarding the control flow within the BT. A node is specialized by its *Component name C*, *Behavior B*, *Type*, and set of *Flags*. The *Behavior* is an identifier (describing a state, an event, or a channel name), or an expression (referring to component attributes).

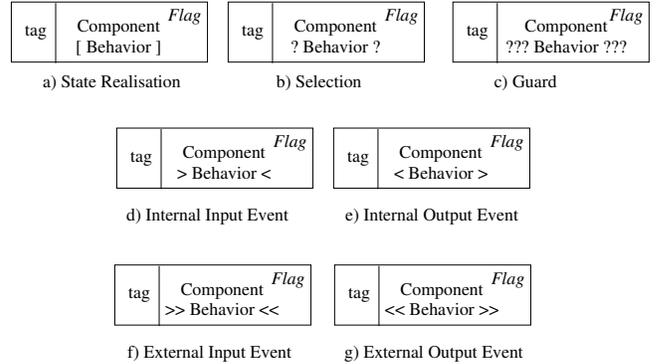


Figure 1. Different node types of the BT syntax

Figure 1 lists the different *types* of BT nodes:

- (a) a state realisation, modelling  $C$  being in a state if  $B$  is a state name, or updating  $C$ 's attribute if  $B$  is an update expression over the attribute;
- (b) a selection (or condition) on  $C$ 's state if  $B$  is a state name, or a selection on the state of one of  $C$ 's attributes if  $B$  is an expression over the attribute; in both cases, the control flow terminates if the condition is not satisfied;
- (c) a guard; the control flow can pass the guard when  $C$  is in state  $B$  if  $B$  is a state name, or when the expression  $B$  over one of  $C$ 's attributes is satisfied if  $B$  is an expression over the attribute; otherwise it is blocked until the state realisation occurs;
- (d-e) an internal event modelling communication and data flow between components within the system, where  $B$  specifies an event; the control flow can pass the internal input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs;
- (f-g) an external event modelling communication and data flow between the system and its environment, where  $B$  specifies an event; the control flow can pass the external input/output event node when the event occurs (the message is sent), otherwise it is blocked until it occurs.

Control flow within the system is represented by either a normal or a branching edge. A normal arrowed edge models *sequential flow* between two steps. If two nodes are

connected by a line without an arrow head, the two steps occur together atomically. Figure 2 shows the two types of branching edges: concurrent and alternative. *Concurrent branching* (Figure 2a) models threads running in parallel. As an example the threads in the figure start with a guard node. The branches, however, can start with any node type. We show only two sub-trees in the branching, although in general there may be more.

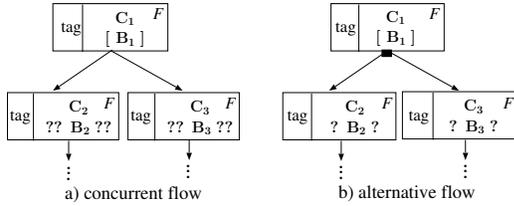


Figure 2. Branching structures in the BT syntax

In *alternative branching* (Figure 2b), the control flow follows only one of the branches. Alternative branches can comprise either selections only (for example, as shown in Figure 2b) or only other node types but no selections. Alternative branching over selections operates as a non-deterministic choice over the branches with a satisfied selection condition  $B_i$ . If none of the selections is satisfied the behaviour terminates. Alternative branching over non-selections behaves like a non-deterministic choice that is unguarded.

*Flags* in a BT node can specify: (a) a *reversion* node, marked by ‘ $\wedge$ ’, if the node is a leaf node, indicating that the control flow loops back to the closest matching ancestor node (a matching node is a node with the same component name, type and behaviour) and all behaviour started after the matching ancestor node is terminated; (b) a *referring* node, marked by ‘ $\Rightarrow$ ’, indicating that the flow continues from the matching node; (c) a *thread kill* node, marked by ‘ $--$ ’, which kills the thread that starts with the matching node, or (d) a *synchronisation* node, marked by ‘ $=$ ’, where the control flow waits until all other threads with a matching synchronisation node have reached the synchronisation point.

The BT syntax also includes notation for standard set operations. For instance, assume  $S$  is a set then the *Behavior*  $S := S + \{x\}$  models adding element  $x$  to  $S$ . Similarly the *Behavior*  $S := S - \{x\}$  models removing element  $x$  from  $S$  (for a complete description of the syntax, including set union, set difference and cardinality, see [29]).

Type declarations and other structural information about the system model are captured in a *Composition Tree* (CT). We do not introduce the notion of CTs here but refer the interested reader to [29]. The semantics of the BT notation is formalised in [11].

## B. Model Checking Behavior Trees

BTs can be interpreted as state transition systems and translated into the syntax of a model checker such as SAL [13] or NuSMV [30]. We have developed an automated translation from BTs to the SAL language which provides an interface to the SAL tool suite. This allows for a fully automated analysis of BT models.

When translating, BT nodes (or a combination of those) are transformed into *actions* in the SAL notation which consist of a guard and an update section. Events like internal or external inputs or outputs are translated into Boolean SAL state variables. External input events in particular become SAL input variables which are not controlled by the model and behave arbitrarily. Boolean variables that represent internal events are set to `true` when the input is sent (by an internal output node) and are set to `false` either if the input is consumed (by the matching internal input node) or if another step has been taken and the event has not been consumed. This reflects that events are fluent and might be missed.

We specify the properties to be checked in linear temporal logic (LTL) [31] and making use of the SAL symbolic model checker. LTL provides a set of temporal operators such as  $G$  (always, in every state of the path),  $F$  (eventually, in some state in the future) and  $X$  (in the next state). Formulas built from these are interpreted over paths and thus referred to as *path formulas*.

To render the transition relation of the SAL model total (a prerequisite for model checking) we add an ELSE action which is enabled if no other action is. That is, all paths in the model are infinite. The notion of paths corresponds to possible *scenarios* of behaviour that the modelled system can exhibit. More detail about the translation is given in [14], [12].

The following patterns are useful for the formalisation of properties that we employed for validation, verification and cutset computation in our approach. Assume  $A$  is a system property (i.e., assertion about the states of the system and its components),  $B$  is a path formula,  $C$  and  $D$  are system components and  $C\_failed$  and  $D\_failed$  specifies their failure.

- $F(A) \Rightarrow B$  says that, in any scenario in which  $A$  eventually becomes true,  $B$  holds.
- $G(A \Rightarrow B)$  says that in every scenario, if a state is reached in which  $A$  is true, then  $B$  holds in the rest of the scenario from that point on.
- $NOT(F(C\_failed \text{ AND } D\_failed))$  holds for exactly those scenarios in which at least one of  $C$  or  $D$  does not fail.

This last property will be used when eliminating cutset  $\{C, D\}$  (for example) in Section VI below.

BTs have no in-built *fairness constraint* to ensure that enabled nodes execute in a “timely” fashion (or even even-

tually) when there are nodes in other threads which are enabled concurrently. This can become an issue when model checking as it often leads to violations of the checked property due to behaviours which are unrealistic or unfair (such as infinite loops in part of the model, with no progress in other parts of the model). For some models we would like to exclude such unfair behaviour. In general this can be done within the model checker (either using the LTL to impose fairness or the fairness assumption of the model checker's modelling notation [32]).

Since a typical scenario of unfair behaviour is that external input dominates other internal behaviour (a BT model can be busy just reading external input events without ever having the chance of (internally) reacting to this) we provide a solution to this problem within our automated translation to SAL. Our translator provides an option for prioritising internal behaviour over behaviour that is triggered by external input. This assumption imposes a form of fairness onto the model in that it excludes the case whereby a process is busy reading constantly changing external input without getting a chance to react.

### III. THE A320 HYDRAULICS SYSTEM – A CASE STUDY

#### A. Subsystems and components

We present the A320 hydraulics system as a case study. This is the hydraulics system of the Airbus A320 aircraft family, which supplies hydraulic power to the aircraft. The case study was originally presented in [8]. An overview of the system can be seen in Figure 3.

There are three kinds of pumps in the system: Electric Motor Pumps which are powered by the electric system, Engine Driven Pumps powered by the two engines of the aircraft and a RAT pump powered by the Ram Air Turbine. The system contains a physical sub-system that is composed of three hydraulic channels (called *h-systems* below) known as Blue, Yellow and Green. The Blue channel is composed of an Electric Motor Pump EMP<sub>b</sub>, a RAT pump and a distribution line dist<sub>b</sub>. The EMP<sub>b</sub> is automatically activated whenever there is at least one engine active. The RAT pump is automatically activated if both engines are lost while the aircraft is flying at a speed of greater than 100 knots.

The Green channel is composed of an Engine Driven Pump EDP<sub>g</sub> and a distribution line dist<sub>g</sub>. The EDP<sub>g</sub> is driven by Engine 1. The Yellow channel is composed of an Engine Driven Pump EDP<sub>y</sub>, an Electric Motor Pump EMP<sub>y</sub> and a distribution line dist<sub>y</sub>. EDP<sub>y</sub> is driven by Engine 2. The pilot activates EMP<sub>y</sub> while the aircraft is on the ground. The two engines are also controlled by the pilot. A Power Transfer Unit, PTU, opens a transmission from the Green system's hydraulic power to the Yellow distribution line and vice versa, as soon as the differential pressure between both systems is higher than a given threshold.

#### B. Safety requirements

As described in [8], the three over-riding qualitative safety requirements for systems on which aircraft depend, such as hydraulics, are:

- 1) at least 3 independent component failures need to occur in order for all h-systems to be lost (a *catastrophic* system failure)
- 2) at least 2 independent component failures need to occur in order for two h-systems to be lost (a *major* system failure)
- 3) at least 1 component failure needs to occur in order for a h-system to be lost (a *minor* system failure)

(The motor industry has similar qualitative safety requirements for software-based systems in cars.)

However, it is not enough to simply prove this is the case. The system designers and safety regulators will also want to know what are the minimal cutsets and co-effectors for the above system failures. That is, precisely which combinations of component failures can lead to system failures, and under what circumstances (such as in what flight modes, or under which operating conditions)? Such information is important for determining whether component failures are truly independent, or whether they could share a common cause.

For the sake of simplicity, the scope of the current investigation is restricted to the departure phase of flight, including on-ground taxiing, takeoff and climb to cruise, but not including later stages of normal flight such as decelerating and landing. We assume that the pilot has turned on both engines and EMP<sub>y</sub>, and exclude pilot error from the current analysis. Nor do we consider and what effect engine activation/deactivation might have on flight (we do however consider its effect on hydraulic power). Finally, we assume that once a component fails it stays failed. These restrictions are made in order to illustrate the approach while avoiding getting bogged down in detail. The model can however easily be extended to cover them, although relaxing some of these assumptions can have significant implications for computational complexity (see Section VII for discussion).

### IV. THE BEHAVIOR TREE MODEL

#### A. The BT model

The BT model of the A320 hydraulic system contains 12 components interacting concurrently: the two Engines (Engine<sub>1</sub>, Engine<sub>2</sub>), the two Engine Driven pumps (EDP<sub>y</sub>, EDP<sub>g</sub>), the two Electric Motor Pumps (EMP<sub>y</sub>, EMP<sub>b</sub>), the Ram Air Turbine Pump (RAT), the 3 hydraulic subsystems (Yellow, Blue, Green), and the Power Transfer Unit (PTU). Additionally, we introduce a component PTU<sub>y</sub> as a modelling artifact representing the OR of the pumps EDP<sub>y</sub> and EMP<sub>y</sub> in Figure 3. Each of these components is modelled as a separate thread in the BT. The Aircraft is also included, in order to model different

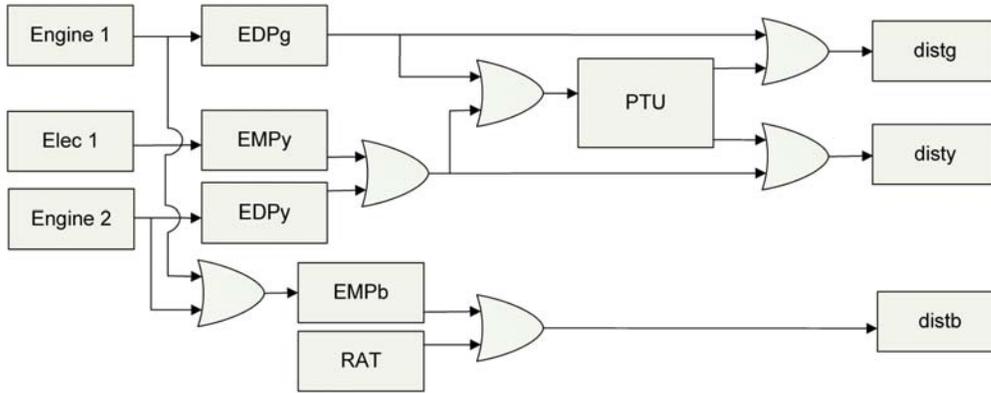


Figure 3. Overview of the A320 Hydraulic System

flight modes. A `System` component is included in order to capture relevant system history information, such as which failures have occurred. Figure 4 shows the overall structure of the BT Model.

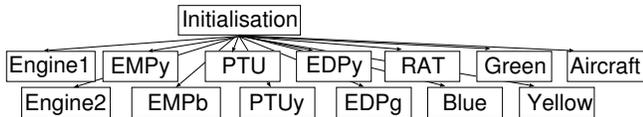


Figure 4. Overview of the BT Model

The system components each have states `active` and `off`, where `off` can represent the fact that it has not yet been activated, or has been deactivated, or has failed. *Active* (/activated) can mean different things, depending on the nature of the component and its uses: e.g., the engine being active means it is able to supply kinetic energy to an EDP pump or electricity to an EMP pump; a distribution line is active if it is capable of transmitting hydraulic power. The activation/deactivation behaviour of each component is modelled by threads. As in [8], we abstract away from lower level h-system components such as tanks, valves and gauges, as well as internal details of the electrical and engine systems.

We initialise `Engine1`, `Engine2` and `EMPy` to `active`, on the assumption that the pilot has followed the standard operating procedure and turned on all three components. (Here we also assume the pilot does not subsequently turn them off.) All other components are initially `off`. The `Aircraft` has three states (`onGround`, `flyingSlow` and `flyingFast`), representing the three flight modes being considered, and is initially `onGround`.

A component failure event is represented by an external input event, since these are outside the control of the system and can happen randomly. The `System` component carries information about which components have failed

(e.g., `distyFailed=true` means `disty` has failed), which h-systems are available (via a set `hset` which can have elements `bh`, `yh` and `gh`, standing for the three h-systems), and how many different components have failed so far (`numFails`).

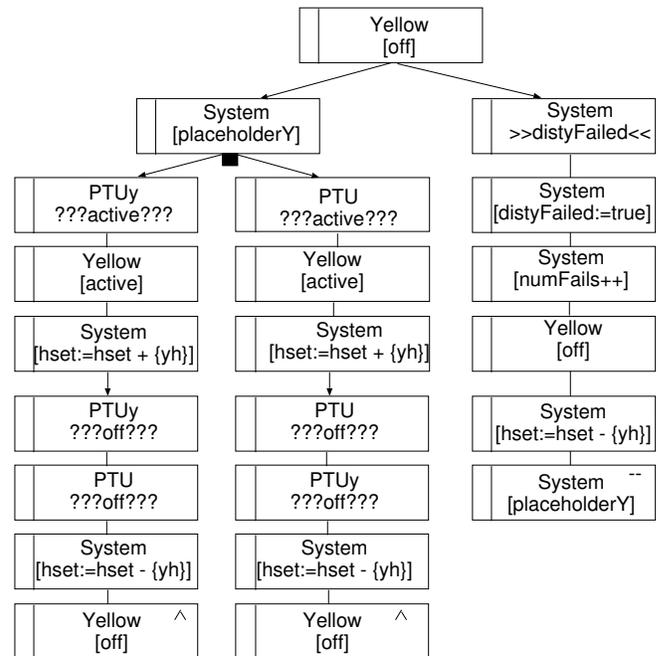


Figure 5. The Yellow subsystem

Figure 5 shows the BT model of the Yellow h-system, which is typical of the style of modelling used here. The left-most thread represents the case where `PTUy` becomes active first, thereby activating `Yellow` and updating the `System` state `hset`. If `PTUy` subsequently deactivates, then we check the other necessary condition for `Yellow` to deactivate, which is that `PTU` is currently `off`; only then do we deactivate `Yellow`

and update `hset`. Reversion is provided in case one of the subsystems subsequently becomes active again. The middle thread represents the analogous case where `PTU` becomes active first. This two-thread pattern is typical of how simple redundancy (components whose functions are OR'ed) can be modelled in BTs.

The rightmost thread represents the case where the `disty` distribution line fails. `Yellow` is set to `off` and the `System` attributes are updated accordingly. Because we only consider permanent failures of components here, `Yellow` will never become active again. We represent this by including a thread-kill node which kills off the other two threads permanently. (A “placeholder” node is used so that the two threads can be killed simultaneously.) The thread then terminates.

The subsystems `Green`, `PTU` and `PTUy`, and pump `EMPb`, are modelled analogously. (In keeping with [8] we have modelled `PTU` as simply transferring hydraulic power between the `Green` and `Yellow` system whenever available, rather than only whenever needed – although it be straightforward to model the latter in BTs.) The pumps `EDPg` and `EDPy` are similar, except that they each depend on only one other component for activation and deactivation (`Engine1` and `Engine2` respectively). Pump `EMPy` normally requires pilot action in order to activate, which is out of scope of the current study, so the model is very simple here, consisting simply of a possible transition to `off` after external input event `>>empyFailed<<`.

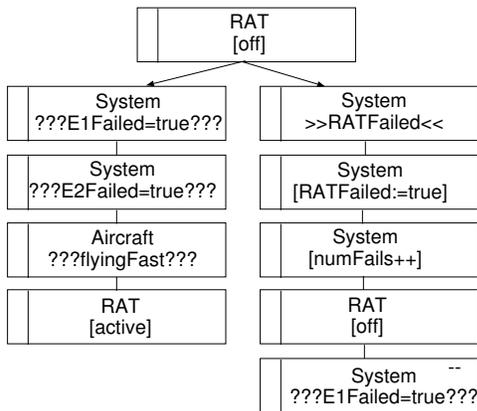


Figure 6. The RAT subsystem

Our model of the `RAT` subsystem is shown in Figure 6. This component gets activated if `Engine1` and `Engine2` have failed and the `Aircraft` is flying fast (i.e., faster than 100 knots) – represented by the thread on the left. Since we do not consider flight phases after `flyingFast` in this study, once activated `RAT` can only be deactivated if the component itself fails – represented by the thread on the right. We have used this fact to simplify the modelling of the `Blue` h-system also (see Figure 7): the left and middle

threads are now parallel threads instead of alternatives; if and when `RAT` becomes active (the middle thread), it overrides (kills) the left thread. The only way that `Blue` can subsequently be deactivated is if `distb` fails.

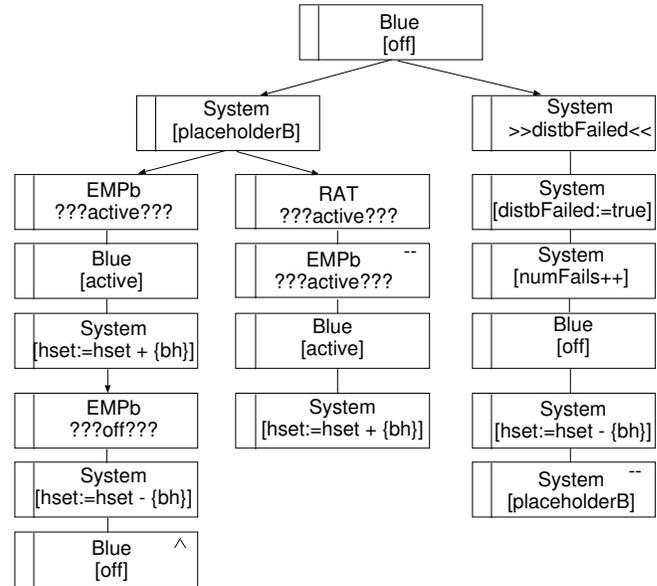


Figure 7. The Blue subsystem

## B. Validation of the Model

We validate our model using the model checker to check properties the system should satisfy. If a violation occurs a counterexample is produced by the model checker that indicates where the model behaves unexpectedly, which helps debugging.

When interpreting counterexamples it is useful to bear in mind that paths in the SAL model correspond to fully played out scenarios in the BT model. Each scenario starts from the root of the BT and follows a thread, taking one of the possible choices at each alternative branching point. Different interleavings of concurrent threads lead to different scenarios. Threads execute either indefinitely in the presence of reversion (leading to a loop in the scenario) or until all threads have terminated (after which only the null ELSE action applies).

As noted in Section II-B we use the BT-to-SAL translator option which prioritises internal actions over external actions. This avoids generation of spurious counterexamples caused by environmental conditions changing before the system has had a chance to react to them, such as e.g. the case where the `Green` system fails before the `Yellow` system has activated via the `PTU`, or where the aircraft transitions to flying fast (thereby activating `RAT`) before other failures have had their effects.

The following definitions will be used below:

- (1) All three h-systems are available:

```
H_ALL:= Yellow=active AND Green=active
      AND Blue=active
or equivalently card(hset)=3, where card refers to
cardinality of the set.
(2) The initialisation period is complete, meaning that the
two engines have delivered sufficient power and current to
activate the various pumps, which in turn has activated all
three h-systems:
INIT:=H_ALL
(3) All hydraulic power is lost:
TOTAL_LOSS:= card(hset)=0
```

In order to validate our model we check for the following two expected behaviours. If standard operating procedures are followed and no components fail, then eventually the system will initialise correctly:

```
SOP1: G(numFails=0 => F(INIT))
```

(Recall that the BT model being checked assumes that standard operating procedures are followed. In Section VII below we relax this assumption.) Moreover, under the above conditions, once the h-systems are activated, they stay active thereafter:

```
SOP2: G(numFails=0 =>G(H_ALL =>X(H_ALL)))
```

The validation process revealed several issues in the model which all have been addressed by modifications. The version described in Section IV-A above is the final, debugged model.

## V. VERIFICATION OF SAFETY REQUIREMENTS

The main safety requirement from Section III is that at least three independent failures are necessary before all hydraulics power is lost:

```
SSR1: G(INIT =>
      G(TOTAL_LOSS => numFails>2))
```

This is essentially the main theorem of Bieber [8]. Note that we are only interested in states of the system after the h-systems have been initialized, hence the use of INIT as an assumption on the left hand side of the implication. (The use of INIT as a proxy for the end of system initialisation is justified by properties SOP1 and SOP2 above.)

The second safety requirement is that at least two independent failures are necessary before only one h-system remains active:

```
SSR2: G(INIT =>
      G(card(hset)=1 => numFails>1))
```

The third safety requirement is essentially a restatement of the SOP1 validation property above, which says that once an h-system becomes active, it will not deactivate unless a component fails. (Recall that we have excluded the case

where the pilot accidentally or purposefully turns an engine off.)

SAL proves all three safety requirements hold for our model.

## VI. USING COUNTEREXAMPLES

In order to generate cutsets, we purposefully ask SAL to check a property which we know not to be true, such as that a major system failure (see Section III-B) is not possible after initialisation:

```
SSR2V: G(INIT => NOT F(card(hset)=1))
```

SAL returns a counterexample revealing a scenario under which this property is false (i.e., in this case, only one h-system is available). Counterexamples are given as a sequence of actions and values for the external variables, possibly including a cycle at the end of the sequence. It is an easy matter to match failure events and co-effectors (such as flight mode changes) with labelled actions in the SAL model, and thus to determine the specific circumstances under which this particular safety violation occurs. This gives us our first cutset, which in the above case is {disty, distb, EDPy}.

At subsequent steps we “eliminate” the cutsets previously obtained, by adding preconditions to the property being checked, using the LTL pattern described in Section II-B above. Thus for example in the above case we next apply SAL to the assertion

```
SSR2Va: NOT(F(distyFailed AND distbFailed
      AND EDPyFailed)) => SSR2V
```

This in turn yields a counterexample with cutset {distb, distg}, which we eliminate by adding NOT(F(distbFailed AND distgFailed)) to the antecedent of SSR2Va. This in turn yields cutset {disty, distg}, which in turn yields cutset {disty, distb} (note that this is a subset of one of the earlier cutsets). Next comes {EMPb, disty}, then {distg, EMPb}, then {disty, EDPg, PTU}, then {PTU, EDPg}, then {E1, disty, PTU}, then {PTU, E1, distb}.

At this point we decided to see if there were any more 2-point cutsets remaining, so we added G(numFails<3) to the antecedent and SAL returned proved. There are thus six 2-point cutsets that can lead to a major system failure. (We note in passing the case where both engines fail while on ground, which leaves EMPy on its own powering both the Yellow and Green h-system (the latter via PTU): while this does not constitute a major failure by the above definition, it is clearly not a desirable situation.)

The counterexamples also yield details of the circumstances (co-effectors) under which the system failure occurs, such as if the engines fail while the aircraft is flying slow (in which case RAT will not activate).

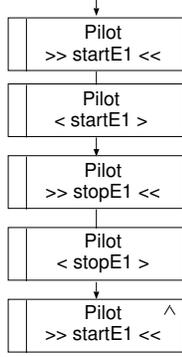


Figure 8. Pilot behaviour: control of engine 1

## VII. DISCUSSION

### A. Including pilot behaviour

For time and space reasons we restricted our analysis of the case study to the case where the pilot follows Standard Operating Procedure (SOP), which includes turning on both engines and EMPy while on the ground, and not subsequently turning them off. One of the advantages of the BT approach is that behaviours of different kinds of components can be represented very naturally, including operator behaviours. In this section we discuss how to extend the model of Section IV-A to include pilot actions and the impact this has on computation time. In the next section we discuss inclusion of pilot error into our analysis.

We extended our model to include a `Pilot` component, with external input events used to model nondeterministic (pilot choice) actions and internal messages to model the effects of pilot actions on the engines and EMPy. Fig. 8 shows the thread corresponding to the pilot turning engine 1 on and off; the model includes similar (parallel) threads for engine 2 and EMPy. Reversion is used because this behaviour can be repeated indefinitely. The `Engine1`, `Engine2` and `EMPy` models get extended with input events corresponding to receiving the pilot commands to activate or deactivate.

In order to examine the effect on computation time we emulated the original model by including `SOP` as a constraint and repeating our analysis of the main safety requirements. Thus for example we checked the property  $SOP \Rightarrow SSR1$  where

$$SOP := F(startE1) \text{ AND } G(\text{NOT } stopE1) \dots$$

The first of the checks (`SSR1`) returned after 7 hours (by comparison with 13 minutes for the original model) whereas `SSR2` did not terminate within 24 hours (by comparison with 65 minutes). This is a clear illustration that it is computationally far more efficient to include constraints directly in the model where possible, rather than as assumptions in the properties being checked. The BT notation's simple integration and structuring mechanisms makes this generally easy to do, and it is often simply a matter of trimming

certain threads and/or removing certain subtrees altogether. Unfortunately current BT tools do not support comparison of different models to see where they differ, which would help make this approach more convenient for model readers and writers, but there are plans to do so in future.

### B. Including pilot error

When the SOP assumption is relaxed and the safety checks are rerun, many situations are revealed where the safety requirements are violated due to pilot actions. Counterexamples are returned relatively quickly (of the order of 5-12 minutes).

For example, the first counterexample returned when `SSR1` is checked is the situation where the pilot has not yet turned on engine 2 and EMPy, and `distb` and `EDPg` fail, leading to total loss of hydraulic power (but only 2 component failures). (Note that turning on only one engine is normally sufficient to activate all three h-systems, which is why it is valid to reuse our formulation of `SSR1` from Section V above.) The first counterexample for `SSR2` is very similar, with the same pilot actions (/inactions) and failure of `EDPg` leading to loss of the Green and Yellow h-systems.

It is also straightforward to investigate more specific scenarios, such as what are the risks if the pilot turns on both engines but forgets to turn on EMPy. This case can be investigated by adding

$$F(startE1) \text{ AND } F(startE2) \\ \text{ AND } G(\text{NOT } startEMPy)$$

to the antecedent. Some of the counterexamples that arise are due to artificial situations such as pilot looping behaviour (such as repeatedly turning an engine on and off) before other subsystems have had a chance to activate. Such problems can be remedied by strengthening the assumptions appropriately, such as by adding

$$FG(\text{NOT } stopE1) \text{ AND } FG(\text{NOT } stopE2)$$

or strengthening it even further to assume SOP has been followed for the two engines at least. In the latter case a counterexample for `SSR1` reveals that all hydraulic power will be lost if both engines fail while on the ground. Refining the analysis further reveals that at least one h-system is still active if the engines fail while in flight, provided there are no other failures. This illustrates the flexibility of the approach in supporting safety assessment, beyond simply verifying safety properties.

### C. Other differences from Altarica model

To a large degree we stayed very close to the Altarica model in [8], since that was the best source of design detail of the A320 hydraulics we could find. However, the authors made some assumptions and simplifications in their model which were not necessary in our model, in part because of the BT notation's expressiveness. For example, we were

able to model the RAT activation's dependence on engine failures simply and directly (see Fig. 6 above). And we could readily have modelled the PTU activation precondition that there should be differential pressure between the Green and Yellow h-systems as a check that both Green and Yellow are not currently active, but chose not to do so for space reasons.

They took a different approach to cutset analysis by including activation conditions as well as failure conditions. We could emulate their analysis by extracting actions corresponding to component activations from the counterexamples, although this would add significantly to the amount of analysis that would be required. They impose "activation properties" as extra conditions in order to emulate SOP and other control actions, whereas we simply represent them as required behaviours within the BT, or as additional assumptions in the LTL formulae we check. These points illustrate the relative expressiveness of the BT notation, over and above how closely it sticks to the natural language specification of the problem.

Finally, they show that the system meets its main safety requirement in a weakened form and through a two-step analysis. More specifically, they need to combine temporal logic reasoning with their model checking results in order to establish that total loss of hydraulic power is temporary (over one time step at worst). By contrast, we are able to state the safety requirement as a simple formula (SSR1) and prove it directly in the model checker. (Their issue arises in part because subsystems may take several steps to activate in their model, with other failures intervening, whereas our model with prioritisation circumvents the problem by not allowing such interventions. A non-prioritised version of the BT model should be used if such interventions are considered plausible.) They did not attempt to verify the other two safety requirements, whereas it was a simple matter for us to do so.

### VIII. SUMMARY AND CONCLUSIONS

The paper demonstrates the use of the Behavior Tree (BT) notation and model checking for assessing system safety. While the overall approach is not particularly new, this is the first reported application of BTs to the treatment of system design complexity resulting from stringent fault-tolerance safety requirements. Verifying the logic of such designs is difficult for humans because of the sheer amount of detail and number of different cases that need to be considered. Our approach automates such analysis.

We demonstrated the approach on a case study from aircraft design: namely, the hydraulics systems of the Airbus A320 family of aircraft [8]. Although relatively small, the example is complex and critical enough to demand very careful analysis. And although many of the components are similar in nature, the precise circumstances under which they are active differs from component to component, and

depends in subtle ways on the mode of operation and on operator actions. We showed that the BT notation yields a simple and concise representation of the configuration logic and activation behaviour, very close in nature to the natural language description of the system. By including appropriate system attributes in the BT, such as means for tracking the number of component failures and the state of top-level functions (in this case, the three independent hydraulic systems), formalisation of safety requirements in temporal logic became simple and transparent.

We also demonstrated a systematic method for using counterexamples to generate cutsets (the combinations of component failures that could lead to hazardous system failures). As illustrations we derived all two-element cutsets that could lead to a major hydraulics system failure, excluding pilot error. We showed that the method applies equally well to systems with iterative behaviour and loops, such as when pilot behaviour is included in the analysis. However these extensions had significant computational overheads.

This work was supported by Linkage Project grant LP0989363 from the Australian Research Council.

### REFERENCES

- [1] Y. Yeh, "Design considerations in Boeing 777 fly-by-wire computers," in *Proc. 3rd Int. High-Assurance Systems Engineering (HASE) Symposium*. IEEE, Nov 1998, pp. 64–72.
- [2] Society for Automotive Engineers, "Certification considerations for highly-integrated or complex aircraft systems," Aerospace Recommended Practice ARP 4754, 1996.
- [3] —, "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment," Aerospace Recommended Practice ARP 4761, 1996.
- [4] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, Feb 2007.
- [5] P. Conmy and J. McDermid, "High level failure analysis for Integrated Modular Avionics," in *Proc. 6th Australian Workshop on Safety Critical Systems and Software (SCS)*. Australian Computer Society, 2001, pp. 13–21.
- [6] R. G. Dromey, "From requirements to design: Formalizing the key steps," in *Proc. 1st Int. Conf. on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2003, pp. 2–13.
- [7] —, "Genetic design: Amplifying our ability to deal with requirements complexity," in *Scenarios: Models, Transformations and Tools*, ser. LNCS, vol. 3466. Springer-Verlag, 2005, pp. 95–108.
- [8] P. Bieber, C. Castel, and C. Seguin, "Combination of fault tree analysis and model checking for safety assessment of complex system," in *Proc. 4th European Dependable Computing Conference (EDCC)*, ser. LNCS, F. Grandoni, Ed., vol. 2485. Springer, 2002, pp. 19–31.

- [9] D. Powell, "Requirements evaluation using Behavior Trees – findings from industry," in *Industry track of Australian Software Engineering Conference (ASWEC)*, 2007, see publications section of [www.behaviorengineering.org](http://www.behaviorengineering.org).
- [10] L. Wen and R. G. Dromey, "From requirements change to design change: A formal path," in *Proc. 2nd Int. Conf. on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2004, pp. 104–113.
- [11] R. Colvin and I. J. Hayes, "A semantics for Behavior Trees," ARC Centre for Complex Systems, ACCS Technical Report ACCS-TR-07-01, April 2007. [Online]. Available: [www.accs.edu.au](http://www.accs.edu.au)
- [12] L. Grunske, K. Winter, and N. Yatapanage, "Defining the abstract syntax of visual languages with advanced graph grammars – a case study based on Behavior Trees," *Journal of Visual Language and Computing*, vol. 19, no. 3, pp. 343–379, 2008.
- [13] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Proc. Int. Conf. on Computer-Aided Verification (CAV 2004)*, ser. LNCS, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 496–500.
- [14] L. Grunske, P. A. Lindsay, N. Yatapanage, and K. Winter, "An automated failure mode and effect analysis based on high-level design specification with Behavior Trees," in *Proc. of Int. Conf. on Integrated Formal Methods (IFM 2005)*, ser. LNCS, J. Romijn, G. Smith, and J. van de Pol, Eds., vol. 3771. Springer, 2005, pp. 129–149.
- [15] N. G. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995.
- [16] P. Fenelon, J. McDermid, M. Nicholson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *ACM Computing Reviews*, vol. 2, no. 1, pp. 21–32, 1994.
- [17] P. A. Lindsay, "Behavior trees: from systems engineering to software engineering," 2010, elsewhere in these Proceedings.
- [18] J. D. Reese and N. G. Leveson, "Software deviation analysis," in *Proc. 19th Int. Conf. on Software Engineering (ICSE)*. ACM Press, 1997, pp. 250–261.
- [19] M. P. E. Heimdahl, Y. Choi, and M. W. Whalen, "Deviation analysis: A new use of model checking," *Automated Software Engineering*, vol. 12, no. 3, pp. 321–347, 2005.
- [20] T. Cichocki and J. Górski, "Formal support for fault modelling and analysis," in *Proc. Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, ser. LNCS, U. Voges, Ed., vol. 2187. Springer, 2001, pp. 190–199.
- [21] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita, "Improving safety assessment of complex systems: An industrial case study," in *Proc. Int. Symp. of Formal Methods Europe (FME)*, ser. LNCS, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805. Springer, 2003, pp. 208–222.
- [22] M. Bozzano and A. Villafiorita, "Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform," in *Proc. Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*, ser. LNCS, vol. 2788. Springer-Verlag, 2003, pp. 49–62.
- [23] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-Simulink models," in *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2001)*. IEEE Computer Society, 2001, pp. 77–82.
- [24] F. Ortmeier and G. Schellhorn, "Formal fault tree analysis – practical experiences," *Electron. Notes Theor. Comput. Sci.*, vol. 185, pp. 139–151, 2007.
- [25] A. Rauzy, "Mode automata and their compilation into fault trees," *Reliability Engineering & System Safety*, vol. 78, no. 1, pp. 1–12, 2002.
- [26] A. Rauzy and Y. Dutuit, "Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia," *Reliability Engineering & System Safety*, vol. 58, no. 2, pp. 127–144, 1997.
- [27] J. Hammarberg and S. Nadjm-Tehrani, "Formal verification of fault tolerance in safety-critical reconfigurable modules," *Int J Softw Tools Technol Transfer*, vol. 7, pp. 268–279, 2005.
- [28] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Proc. 32nd ACM/IEEE Design Automation Conference (DAC)*. ACM, 1995, pp. 427–432.
- [29] BE. (2010) Behavior Engineering website. [Online]. Available: [www.behaviorengineering.org](http://www.behaviorengineering.org)
- [30] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *Proc. Int. Conf. on Computer Aided Verification (CAV)*, ser. LNCS, vol. 1633. Springer-Verlag, 1999, pp. 495–499.
- [31] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science Publishers, 1990, vol. B.
- [32] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2000.