Model-based Safety Risk Assessment using Behaviour Trees

Peter A. Lindsay[1,2], Kirsten Winter[1] and Sentot Kromodimoeljo[1]

[1] School of Information Technology & Electrical Engineering,
The University of Queensland, Brisbane, Australia

[2] National ICT Australia, St Lucia, Qld

E-mail: p.lindsay@uq.edu.au

Abstract. For complex engineered systems, it is important to conduct technical risk assessment early in the system development life-cycle, in order to identify critical system requirements, such as safety requirements, that should be included in design. This paper proposes a model-based approach to such assessment, which can be applied from the system requirements analysis stage onwards.

The approach starts with the application of the Behaviour Trees modelling notation to natural language functional requirements. The BT model is then extended to include the events and conditions that might contribute to hazards, and automated model checking is used to identify the mechanisms by which component or subsystem failures can lead to hazardous system failures. The approach is intended to be used iteratively in design and analysis, to assist system designers in assessing the effectiveness of system safety requirements. A hypothetical bushfire-fighting management system is used to illustrate the approach.

1. INTRODUCTION

Overview and Scope

This paper introduces a formal process to assist system analysts to identify system safety requirements, and check their effectiveness, early in the system development life-cycle, on models derived directly from natural language Functional Specifications. The approach uses the Behaviour Tree (BT) notation (Dromey 2003) to model system requirements and automated model checking to check the effectiveness of the proposed safety features.

Behaviour Trees use a graphical notation that has been shown to be easy to understand by people who are not formal methods experts (Powell 2010). The BT notation is distinguished from other formal modelling notations by its ability to capture functions, object states and multi-threaded behaviour in a single modelling language (Lindsay 2010).

Although the paper's emphasis is on system safety, the approach is applicable to any risks that concern system functionality and behaviour, including risks to security, reliability, maintainability, and other system "-ilities". We coin the term *functional risk* (by analogy to functional safety) to mean system risk associated with what functions the system and its components perform – or do not perform – and in what order, both when "all is well" and when one or more of these functions has failed. For example, in the case study below, system functions include storing fire-fighting mission data and calculating when and where to drop fire retardants. Safety risks associated with the system include dropping retardants on the wrong place and dropping retardants inadvertently.

The approach can be used on sets of functional requirements of arbitrary detail – whether it is very early in the life-cycle when functions are defined only in broad abstract terms, or later when a detailed system functional architecture is available. No assumptions are made about how the functionality is implemented: it could be by hardware, software or human operators, or some combination of these, or

it need not even have been decided yet. At this stage the approach is restricted to purely functional behaviour (such as the order in which system functions are invoked and the conditions under which they are invoked): performance issues – including real-time timing issues – and data correctness are not currently covered.

We assume that the main system hazards have already been identified: that is, the states that the system and its functional components could get into that might lead to harm or other undesired consequences. We adapt the BT method of developing models to include component failures and other events and conditions that might contribute to hazards. We then use automated model checking to identify the system behaviours that lead to hazardous system failures.

We contend that this process leads to better understanding of functional risk and helps system designers to identify safety requirements for preventing or mitigating the hazards. The approach is intended to be used iteratively, to assist designers in assessing the effectiveness of system safety requirements and in assessing residual risk. The approach can be also used to improve understanding of the consequences of functional requirements not being satisfied – whether the reason is component failure-in-operation, design failure, or missing functionality (for example, in preliminary builds in a spiral development, or while functions are unavailable during maintenance).

Innovation and Importance

Many other methods of system modelling and analysis have been developed, but typically using specialist notations with difficult semantics. This makes them difficult to integrate into more general systems engineering processes, which in turn means they quickly become sidelined as system requirements change. Also, many of them do not scale up well, to handle analysis of large requirements sets (Broy 2011). By contrast, our approach stays close to the natural language requirements through the use of the BT notation, and scales well to handle very large Functional Specifications. We contend that these advantages lead to improved understanding of system requirements by all stakeholders, and closer integration of safety analysis into standard systems engineering processes.

The main innovation in this paper is to demonstrate how to integrate the results of Preliminary Hazard Identification (PHI) into BT models and then use automated model checking to help identify functional risk. Automation means that analyses can be repeated quickly after changes are made to requirements.

Traditionally technical risk assessment is not performed until the system architecture and design are well advanced. Many experts have pointed out however that system developers need to start considering safety aspects early in development, to ensure that adequate safeguards are incorporated into the system concept, architecture and design. Leveson points out for example that it is better to build safety into a system rather than to try to add it on later (Leveson 1995). Techniques such as Functional Failure Analysis can be applied early but lack tool support. This is the gap that our approach addresses.

There are many other reasons for conducting functional risk assessment early in development. For engineering organisations tendering for fixed-price contracts, it is particularly important to understand technical risks prior to tendering, since managing them can have a significant impact on cost. It is also important during planning if development is to be done incrementally (such as when undertaking spiral development), in order to decide which areas of functionality to include in different builds. Finally, it is important for the organization(s) which operate and maintain the system, so that they can develop Standard Operating Procedures that ensure safety is maintained during operation, and fallback procedures in the event of system failures.

Outline of the Paper

This paper reports the results of an application of the approach to a large case study supplied by Raytheon. The case study concerns a hypothetical system of systems for managing bushfire fighting. The challenge was to extract functional requirements for a particular system – an Aerial Fire-fighting Management System (AFMS) – from a Functional Performance Specification consisting of thousands of requirements, and then to identify some of the system hazards that could arise, use the approach to help identify the system behaviours that could give rise to such hazards, and identify the risk remaining after safety features were added to the system design. The preliminary results on which this paper is based are available on the web at http://itee.uq.edu.au/~dccs/AFMS.

Section 2 introduces the BT method and notation and the translation to the SAL model checker. Section 3 summarises the proposed approach to supporting functional risk assessment. Section 4 describes the AFMS case study. Section 5 describes a BT model developed from the requirements using the BT method. Section 6 describes the AFMS hazards that were chosen for study and shows how to adapt the BT model to include component failures and safety requirements. Section 7 illustrates the use of the SAL model checker to reveal behaviours leading to system hazards. Section 8 repeats the analysis after certain safety requirements have been added to the system, to show that risk has been reduced effectively. Section 9 describes related work and Section 10 describes conclusions and future work.

2. BACKGROUND

The Behaviour Tree Notation

Our approach is based on the use of the Behaviour Tree (BT) notation for capturing requirements that are given in natural language (Dromey 2003).

The strength of the BT approach is two-fold: Firstly, the graphical nature of the notation provides the user with an intuitive understanding of a BT model - an important factor especially for use in industry. Secondly, the process of capturing requirements is performed in a stepwise fashion. That is, single requirements are modeled as single BTs, called individual requirements trees. In a second step these individual requirement trees are composed into one BT, called the integrated requirements tree. Composition of requirements trees is done on the graphical level: an individual requirements tree is *merged* with a second tree (which can be another individual requirements tree or an already integrated tree) if its root node matches one of the nodes of the second tree. Intuitively, this merging step is based on the matching node providing the point at which the preconditions of the merged requirement trees are satisfied. This structured process provides a successful solution for handling very large requirements specifications (Powell 2010).

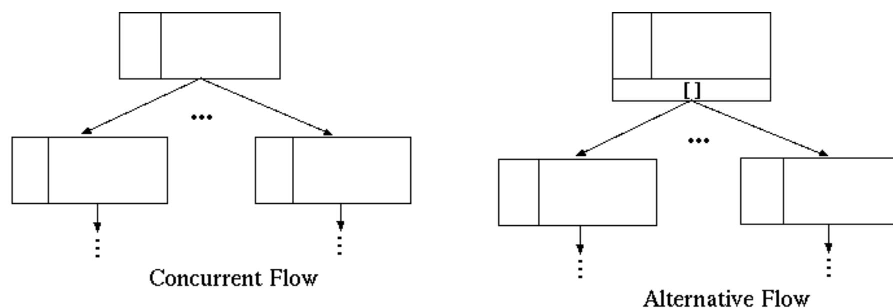

Figure 1: Parallel vs alternative branching syntax

The BT notation is a graphical modelling notation which represents a system's behaviour as a tree-like structure consisting of nodes and edges. Flow of control starts at the root of the tree and follows paths down the tree, with nodes representing component state changes and message dispatch and receipt. (In this paper "component" can mean a group of functions a logical architecture as well as a subsystem or physical component, including human operators.) There are two kinds of branching:

parallel branching (LHS of Fig.1) whereby a new thread of control is forked for each branch; and alternative branching (RHS of Fig.1) in which flow continues down at most one of the branches. Generally threads are independent and proceed concurrently "at their own pace", subject to synchronization and other flow directives as explained below. The exception to this is when multiple nodes are linked *atomically* as a block with no edges between them: when flow reaches such a block, all other threads wait until all of the nodes in the block have been "executed".

The different kinds of BT nodes are shown in Fig. 2. A component might change its state (*state realisation*), receive an input from another component (*internal input event*) or from the environment (*external input event*) or similarly, send an output to another component (*internal output event*) or to the environment (external output event). A special case of a state realization is attribute assignment, written *attrib:=value*, whereby the value of one of the component's attributes is modified. If an input is not available when control flow reaches an input event node, the system has to wait until the event is received. Similarly, when control flow reaches a guard node, the system waits until the guard condition becomes true before proceeding. If flow reaches a selection node, it continues on if the selection condition is true and terminates if not.

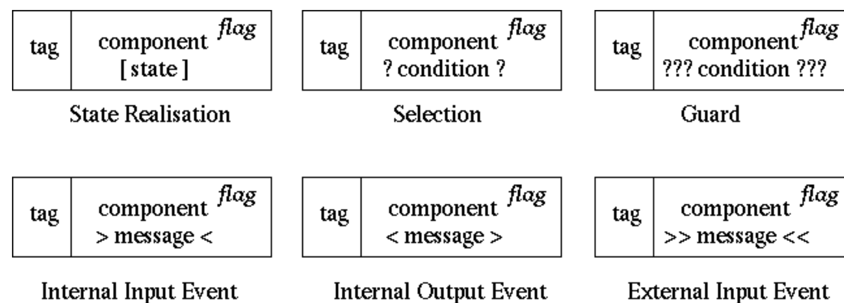

Figure 2: Syntax of Behaviour Tree nodes

As shown in Fig. 2, a node may also contain tags and/or a flag. Tags are used for tracing a (set of) node(s) to the requirement(s) that gave rise to them. Flags are control flow directives: When flow reaches a leaf node with a *reference* flag (=>), control jumps to a matching node: this can be either an explicitly named node, or – provided there is no ambiguity – a node elsewhere in the tree with the same contents; this is essentially a device to avoid duplicating sub-trees in the model. When flow reaches a leaf node with a *reversion* flag (^) flow jumps back up to a matching node above the current node; moreover any other threads of control that started below the target node get terminated. The BT notation also has flags for killing sibling threads and for synchronizing threads, but they are not used in this paper.

The BT notation includes other operators and node types that are not introduced here. For a full description of the syntax the reader is referred to the Behaviour Engineering web site[1]. The operational semantics, which is informally described above, is fully formalized in (Colvin & Hayes 2011). The BT notation supports relationships (the what, why, where, etc of requirements) but at this stage the SAL translation is fairly rudimentary and is restricted to data relations.

Model Checking Behaviour Trees

BTs can be interpreted as state transition systems and translated into the syntax of a model checker such as SAL (de Moura et al. 2004)[2] or NuSMV (Cimatti et al. 1999). The translation to SAL is fully automated and has been described elsewhere (Grunske 2005) and SAL is freely available. But in principle other model checkers could be used.

As input the model checker takes a model, which in our case is a BT model translated into SAL code, and a property to be checked. Paths through the SAL state model correspond one-to-one with system

---

[1] www.behaviourengineering.org
[2] http://sal.csl.sri.com/

behaviours in the BT model, in the sense of instances of control flow in interleaved concurrent threads as allowed by the BT model semantics given above. SAL is able to check properties stated in linear temporal logic (LTL) (Emerson 1990), which is a rich language for talking about the order in which things happen (or don't happen). In Sections 6-8 below we make use of this capability to formulate functional safety concerns – such as system hazards – as LTL formulae.

LTL provides temporal operators to express assertions about paths through the SAL model: *G(P)* means that a proposition *P* holds *generally* – i.e., at every step on the path; *F(P)* means that *P* holds *eventually* on the path**;** *X(P)* means that *P* holds at the next step on the path; and *P U Q* means that *P* holds until *Q* holds (and *Q* does eventually hold). [3] Formulae can be built using standard propositional connectives (and, not, implies, etc). For the most part we use only G and standard propositional logic below. Finally, atomic formulae correspond to statements about what state a component is in, the current value of an attribute, or whether a particular message is currently available.

When given a model and a property, SAL either returns *proved* (meaning it has shown that the property holds true on all paths through the model), times out (if it runs out of computing resources), or returns a counterexample. A *counterexample* is a path through the SAL model for which the property does not hold. It is a simple matter to trace a counterexample back to a sequence of steps through the BT model illustrating a system behavior which violates the property. The path need not be continuous, since different threads can progress at different rates, but in keeping with BT semantics it will represent one of the behaviours that is allowed under the (BT model of the) requirements. In cases where the BT model has not constrained initial values of states or attributes, SAL is free to choose its own values. At each step SAL can also choose whether or not a particular external input message is available. This ensures that all possible scenarios are taken into account.

Although SAL only returns a single counterexample at a time, it is usually possible – by further constraining the property to be proved – to eliminate from consideration the particular conditions that gave rise to the counterexample, and then use SAL to find more counterexamples (if they exist). This process is illustrated in Section 7 below.

In this paper we use the "prioritised" version of translation to SAL, whereby internal steps get priority over external events. That is, when control flow reaches an external input message it waits there until all other threads have finished or reached an external input message. (We take care to ensure that every loop in the BT model contains at least one external input message, to avoid live-lock. This sometimes entails inserting a dummy node.) Using priortisation significantly speeds up model checking while at the same time eliminating many spurious counterexamples. In effect we are assuming – for the purposes of analysis – that internal system processes run faster than the environment is throwing new events at the system, which is a reasonable assumption for our case study. But the analyst can relax this assumption by inserting dummy nodes at suitable "break points" or alternatively simply use the non-prioritised version of translation.

3. THE FUNCTIONAL RISK ASSESSMENT PROCESS

In outline, the proposed process is as follows:

1. Start with a set of natural-language functional requirements for the system.
2. Conduct a Preliminary Hazard Identification (PHI) and determine which functional failures and system hazards are of interest.
3. Develop an initial BT model from the functional requirements at a suitable level of detail.
4. Adapt the BT model to include the functional failures of interest and to make it suitable for model checking: see below for more detail.
5. Express the system hazards as temporal logic properties and use the model checker to check if they are reachable in the BT model.

[3] http://sal-wiki.csl.sri.com/index.php/FAQ#Semantics_of_the_LTL_operators

6. If hazards are reachable, check the counterexamples returned by the model checker to determine which behaviours give rise to them. If the risk that such behaviours could occur is unacceptable, add safety functions to the system design to try to eliminate the hazards or mitigate the risk.
7. Repeat steps 5 and 6 until the remaining risk is acceptable.

The process is illustrated on a case study below.

4. THE AFMS CASE STUDY

The case study concerns a hypothetical system for command and control of bushfire fighting from aerial vehicles, called the *Aerial Fire-fighting Management System (AFMS).* The AFMS is part of a larger system of systems called the Bushfire Fighting Management System (BFMS), which coordinates fire fighting across a coalition of different fire-fighting and emergency service organisations.

The main task of the AFMS is to manage aerial fire-fighting missions. Different missions can involve different types of fire retardants being dropped in different kinds of patterns – collectively called *drop solutions* here. The AFMS is concerned with collecting and collating the information that is required to calculate a drop solution, and with passing drop-solution instructions to the drop system on the aircraft - called the drop bay here. The focus in this case study is on the on-board component of the AFMS, called the *Aerial Fire-fighting Control System* (AFCS). The BFMS Command and Control (C2) units and other BFMS units will also contain AFMS components but – apart from their logical interfaces with AFCS – these are not being considered here. Figure 3 shows the logical architecture of the AFMS.

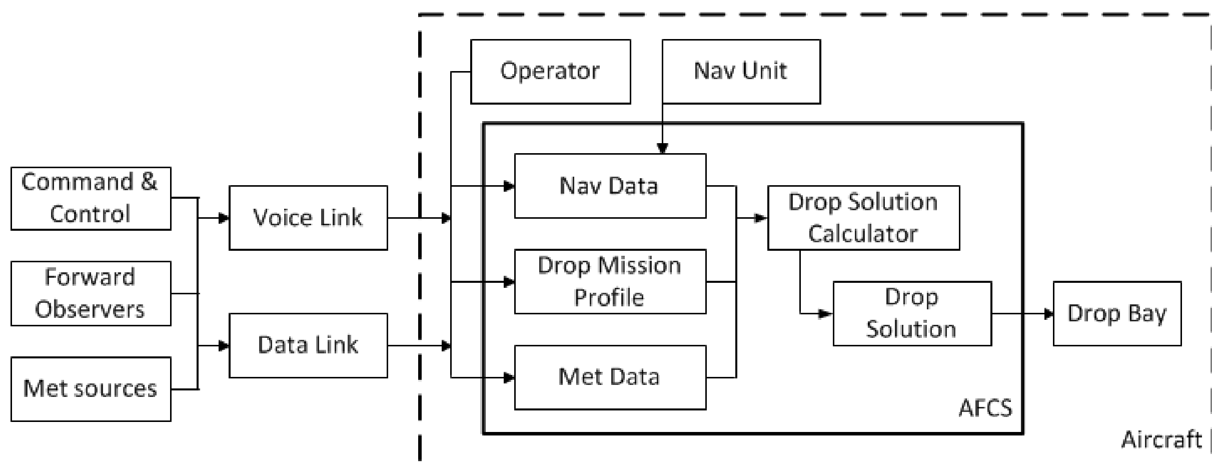

Figure 3: AFMS Logical Architecture

In Figure 3, a solid box labeled AFCS represents the boundary for the main focus of our case study. The AFCS interacts via voice link or data link with Command and Control (C2), Forward Observers (FO) and Meteorology (Met) data sources. The AFCS itself physically resides on an aircraft (represented by a dashed box in figure 3) and also interacts with an operator, a Navigation (Nav) system and a Drop bay, all of which are aboard the aircraft. Pieces of information that are needed to calculate a drop solution include:

- Drop Mission Profile (DMP): the main piece of information specifying mission data such as target, retardant type, volume, and (for scheduled missions) time of drop.
- Navigation (Nav) data: indicating the position, heading and speed of the aircraft.
- Meteorology (Met) data which include temperature and wind conditions.

The Drop Solution Calculator is an off-the-shelf component that produces a drop solution for the drop bay to execute.

Requirements

The requirements for the AFMS are given in a natural language functional performance specification (FPS). There are 58 requirements in the FPS for the AFMS taken from a larger set of requirements (approximately 1700 requirements) for the BFMS. We will not give the entire set of requirements for the AFMS in this paper. The following key requirement gives a flavour of the natural language requirements:

AFCS-51 The AFCS shall be able to automatically carry out the following sequence:

a. receive a call for drop/drop mission from the voice link or data link;
b. calculate the drop solution;
c. prepare drop (for those variants with automated drop mechanisms only); and
d. display the drop solution to the user for confirmation and execution.

The set of requirements in its entirety is available on the web at http://itee.uq.edu.au/~dccs/AFMS/.

5. INITIAL MODEL

Additional Client Information

We constructed an initial model of the AFCS using the BT notation which we called BT Model 1. The initial model tries to faithfully capture the requirements and tries to avoid the forcing of design decisions as much as possible. However, we have added into our model the handling of 3 different types of missions based on how they are initiated: scheduled, operator initiated or on call. The 3 different types of missions were not mentioned in the requirements but were suggested by our client. We have added them to our model because of the possibility of risks specific to how a mission is initiated.

Simplifications for Modelling and Analysis

While the addition of 3 different types of missions adds complexity to the model, we have also taken steps to simplify the model to an appropriate level of abstraction for our analysis:

- FOs are not in our model since their behaviour is indistinguishable from C2 for our purposes.
- C2 is not modelled in detail beyond the ability to send Drop Mission Profiles (DMPs) and other messages.
- Internal details of the Navigation System and Data Link communications are omitted.
- External sources of Met Data have been reduced to C2.
- Although implicitly the different fields in a DMP can be modified individually, in the model we simply pass or modify whole DMPs.
- We focus on risks associated with a single DMP only. In our model, the AFCS stores only one DMP at a time.
- Situational awareness data (mentioned in the requirements) are omitted.
- Details of inputs required for calculation of the Drop Solution have been reduced to the DMP, Met Data and Nav Data.

Note however that such details could be modelled in BT notation if desired. (Modelling of the handling of multiple DMPs requires the use of parameters and quantification in BTs, which were not explained in Section 2 above.)

Overview of the Initial Model

The initial model of the AFMS has 4 main threads:

- A thread that handles the updating of Met Data.
- A thread that handles the updating of Nav Data.
- A thread that handles the management of the DMP, which itself has 3 sub-threads: for DMP reception from C2, for DMP creation by the operator, and for DMP modification by the operator.
- A thread for preparing and executing a drop solution (which consumes the DMP).

In the requirements, data can be sent from C2 to the AFCS via a data link or a voice link. Although not specified in the requirements, we have interpreted this to mean that the data link is the preferred channel of communication, with the voice link acting as a backup. This is reflected in our model in the sending of Met Data from C2 to the AFCS and in the sending of a DMP from C2 to the AFCS.

Figure 4 shows the fragment of the BT for our initial model that deals with the sending of a DMP from C2 to the AFCS. There is an alternative branching at node 6.1. If the data link is down (represented by component *DataLink* having *down* as the value of its attribute *status* here), then the branch headed by selection node 6.2 is taken: the DMP is sent via the voice link, the operator then has to explicitly enter the details of the communicated DMP into the AFCS (represented by the state *updateDMP*), and the DMP status is set to *ready*. If the data link is up, then the DMP is sent via the data link, and there is no operator involvement. In either case, the thread reverts back to the node *ManageDMP[start]*, which is at the start of the thread that handles the creation and updating of DMPs (the third thread).

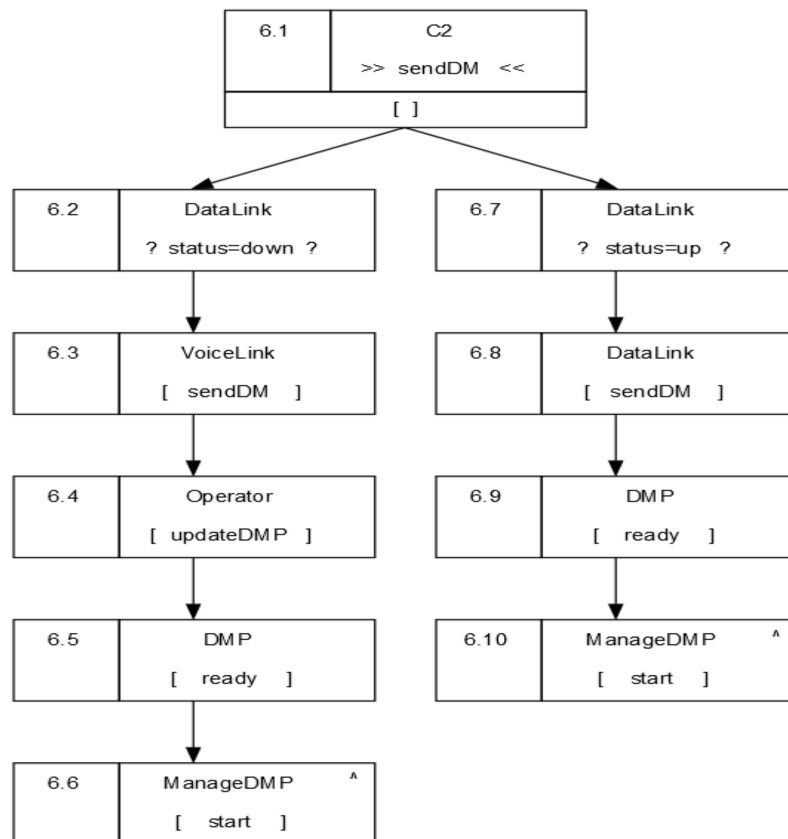

Figure 4: Sending of a DMP from C2 to the AFCS

Figure 5 shows fragments of the BT for the AFCS thread corresponding to preparation and execution of a drop solution for a scheduled mission. The thread first waits until the condition that DMP is *NOT(none)* is satisfied: i.e., a DMP has been received or entered into AFCS by the operator. After the condition is satisfied, then there are 3 alternative branches corresponding to the 3 ways that a drop mission can be initiated. Only the case where the drop mission is scheduled is shown. An operator-initiated mission is handled similarly but with the operator entering an execute command acting as the trigger instead of having a time trigger. An on-call mission is slightly more involved because the trigger is sent by C2 and may go through the data link or the voice link depending on the status of the data link. All 3 cases have nodes 9.4 through 9.14 in common. Nodes 9.4 through 9.8 represent the preparation of the drop solution. After the drop solution has been prepared, the operator has to confirm the drop solution. Once confirmed, the drop solution is sent to the drop bay, the states of several components are modified, and the thread reverts to node 9, all within a single atomic action.

Note that in the BT model, we introduced auxiliary functional components that are not in the AFMS logical architecture diagram. For example, the BT fragments in Fig. 5 contain the auxiliary components *ExecuteDM* and *PrepareDS*. Some of these auxiliary components were created as targets of reversions and references, e.g., *ExecuteDM*. Others were created to indicate the stages or statuses of processes, e.g., *PrepareDS*. The use of auxiliary components is often important in enabling the specification of temporal (LTL) properties and in interpreting model checking results.

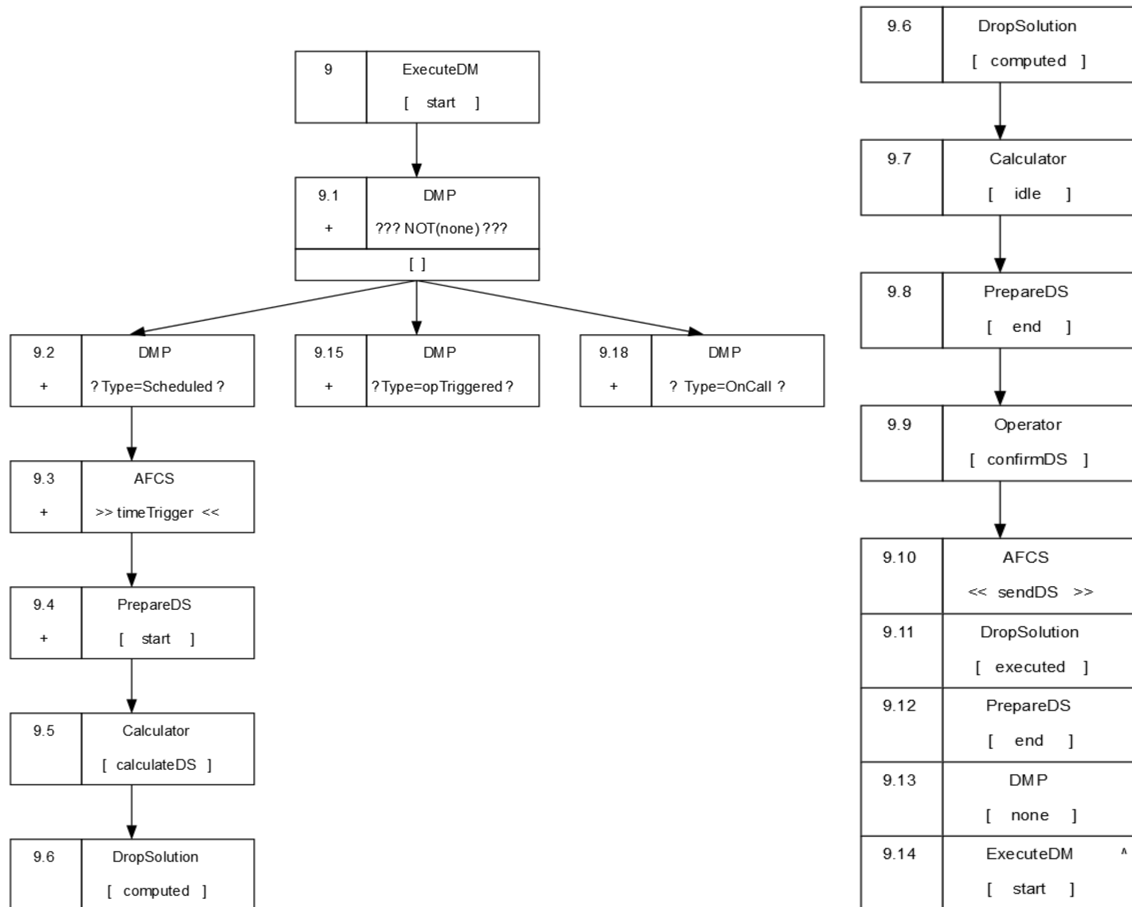

Figure 5: Preparing and Executing a Drop Solution for a scheduled mission

We have not shown portions of the BT for Met Data update, Nav Data update and the rest of DMP management. The BT in its entirety is available on the web at http://itee.uq.edu.au/~dccs/AFMS/.

6. PRELIMINARY HAZARD IDENTIFICATION AND MODELLING

The PHI identified the following main types of hazardous incidents associated with the AFMS:

- Retardants are dropped on people or areas where they should not be
- An inappropriate type of retardant is dropped
- Retardants are dropped inadvertently

To a large degree the system failures that could give rise to such incidents relate to the details that get put into the DMP (data correctness and data integrity issues). Another class of system failures relate to timing issues, such as the drop mission being carried out too early or too late. The system functional failures that give rise to such incidents were of two main types:

- Inputs to the drop solution were out of date
- The DMP was incomplete at the time the drop solution was calculated

With respect to “reasonably foreseeable circumstances”, the following component failure modes were

considered most likely:

- The data link is down
- The interface to the Nav unit fails
- The operator fails to enter data in a timely manner

To enable automated analysis we extended the initial BT model to cover the events and conditions of interest for safety analysis. This included adding behaviours corresponding to the data link going down and the Nav system failing. (For simplicity we considered only irrecoverable failure in both cases, although in principle other failure modes could be included.) We also converted all operator actions *Operator[doesX]* to external input messages of the form *Operator>>doesX<<* so that the model checker would check all possible orderings of operator actions. (This is simply a modelling trick, without functional implications.)

Because BT models do not handle timing performance, we had to address the out-of-date calculator inputs failure indirectly. Upon closer investigation it became apparent that the main way that such failures could arise because of AFCS-related behaviour would be in the case where the data link is down and the system is relying on the operator to enter the details. To address this we introduced notion of data being “pending”. This corresponds to the state of the system whereby the operator has received new data by voice link but has not yet entered it into the AFCS. At this point no new functionality is implied: the new states *MetData[pending]* and *NavData[pending]* are simply placeholders for point in the update-data threads of behaviour which we want to refer to in analysis.

To address the system failure whereby a solution is calculated based on an incomplete DMP, we introduced the notion of a DMP being “ready” or “not ready”, and changed the guard in the *ExecuteDM* thread to wait until the DMP is ready before enabling execution. In effect, we are adding a set of safety requirements to the AFCS along the following lines:

> ***AFCS safety requirement SR-1****: The AFCS shall store the status of the DMP as being either ready or not-ready. This field shall be updatable by C2 automatically over the data link. The operator shall be able to edit the field in the AFCS. The drop solution execution sequence shall not proceed until the DMP has ready status.*

In the original model the operator needed to confirm a drop solution before it got passed to the Drop Bay, but the requirements said nothing about what would happen if the drop solution was not satisfactory. We assumed that the effectiveness of this confirmation step would be enhanced from a safety perspective if the operator had the option of modifying the DMP (or waiting until other input data had been updated) before re-invoking the *PrepareDS* functionality, so added a branching behaviour after rejection.

The resulting model is called Model 2 on the web. The new nodes are identified in white.

Hazard Formulation

We chose two particular hazards to illustrate our approach in this paper. The initial properties to be checked are as follows:

H1 The drop solution is calculated with out-of-date met data.

This is formulated as the condition that the Calculator component is in state *CalculateDS* while MetData is in state *pending*. (The formalization and treatment of the out-of-date nav data hazard is very similar and so has been omitted for reasons of space.)

H2 The drop solution is calculated before the Drop Mission Profile is ready:

This is formulated as the condition that the Calculator component is in state *CalculateDS* while DMP is in state *notReady*.

7. FIRST ROUND OF MODEL CHECKING RESULTS

This subsection reports results of model checking using the prioritized version of the translation of the model above. Full results are available on the web at http://itee.uq.edu.au/~dccs/AFMS/.

H1 Analysis Results and Derived Safety Requirements

To check whether hazard H1 is reachable we first try to prove *G(NOT(H1))*.

SAL finds the following counterexample: SAL starts by assuming the data link is down, and then selects a sequence of steps corresponding to the operator creating a DMP and setting it to *ready*. SAL then selects the operator-triggered DM case under *ExecuteDM*. In parallel under the *UpdateMetData* area of functionality it sets up the conditions for *MetData* to reach the state *pending*. It does this by having C2 send new Met Data, which is received over the voice link but not immediately entered into AFCS by the operator. Instead the operator triggers *PrepareDS* by entering the execution command, which leads in turn to the Calculator being invoked, even though met data is still pending (the hazardous condition!).

The analyst would step through the counterexample and reflect on how likely such behavior would be. In fact this particular case would be relatively unlikely to occur if the operator is well trained, since it requires the operator to trigger DS preparation manually while knowing that met data needs updating. (We are assuming the AFCS has a single operator.) Such a situation could largely be prevented by training the operator of the importance of having up-to-date met data before triggering DS preparation. This requirement should be written into SOPs and training manuals.

To check if there are other system behaviours that could lead to H1 we can eliminate the previous case from consideration by adding to the property to be checked the constraint that the operator does not trigger DS preparation. The new property to model check is *G(NOT(P)) => G(NOT(H1)),* where *P* represents the event *enterExCmd*.

Now SAL returns a counterexample consisting of a scheduled DM which gets triggered by the AFCS clock while met data is still pending. This case is much more likely than the previous case. In Section 8 below we propose safety requirements to addresses this risk and use model checking to evaluate their effectiveness form a functional risk viewpoint.

To eliminate this second case from consideration and continue to investigate other ways the systems hazard could arise, it became apparent that our original formulation of the H1 was too simplistic: SAL kept coming back with examples where the Calculator got invoked but new met data arrived later. We remedied this by reformulating H1 more specifically as being that met data is pending at the time when the Calculator gets invoked. This can be formulated in LTL as follows:

NOT(Calculator = calculateDS) AND X(calculator = calculateDS) AND metData = pending

The analysis reveals some other interesting cases, such as a loop where the operator keeps rejecting the drop solution while met data is pending. But we stopped the analysis once we knew enough about how H1 can arise to propose a suitable prevention mechanism (see Section 8).

H2 Analysis Results and Derived Safety Requirements

The analysis of H2 proceeds in a similar fashion. Upon being given *G(NOT(H2))* to check, SAL returns a counterexample very similar to the first one returned for H1 above, except in this case the operator modifies the DMP after *ExecuteDM* has been invoked but before the Calculator is invoked.

This is an example of a whole group of cases whereby DS calculation gets triggered without operator intervention, while the operator is making some last minute adjustments to the DMP. In Section 8 below we investigate the addition of a safety feature whereby AFCS checks again that DMP is still ready before triggering the Calculator.

8. RE-ASSESSMENT AFTER ADDING SAFETY REQUIREMENTS

In the previous section we used the insights derived from model checking to identify mechanisms by which the hazards could arise in Model 2. In this section we propose safety features that could be added to AFCS to try to eliminate such behaviours, and then reapply the analysis to check what effect they have on functional risk.

To address the first case, where the operator triggers drop solution preparation while met data is pending, we propose adding system safety requirements along the following lines:

> ***AFCS safety requirement SR-2****: Before a drop solution is calculated, if the data link has been down for some time, the AFCS shall prompt the operator to confirm that any pending met data updates have been entered into the system.*

See Fig. 6 below for the behaviours that were added to the BT model corresponding to SR-2.

To address the second case, where the DMP is in the process of being modified after once having been ready but is not currently ready to be implemented, we propose adding a new safety feature along the following lines:

> ***AFCS safety requirement SR-3****: Immediately before a drop solution is calculated, the AFCS shall recheck that the DMP status is still "ready". If not, the execution sequence shall be interrupted until the DMP status is "ready" again.*

The result of adding SR-2 and SR-3 to the BT model is given as Model 3 on the web.

When SAL is applied to the revised model, both of the properties come back as *proved*. This shows that, at this level of abstraction, much of the risk has been eliminated by the new safety requirements.

We undertook some experiments using the non-prioritised version of SAL translation. In this case the model checker returned counterexamples corresponding to race conditions, such as where new met data arrives over the voice link just after the operator has confirmed there is no pending met data and just before the Calculator starts. (Recall that in the prioritized version such behaviour is not considered, since AFCS internal action get priority over external events.) Such "windows of opportunity" are relatively small and could be dealt with by building safety margins into calculations (a data issue rather than a functional issue, in our terms). We leave it for future work to examine these cases more closely.

9. 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 (Grunske et al. 2005), (Reese and Leveson 1997), (Heimdahl et al. 2005), (Cichocki et al. 2001), (Bozzano and Villafiorita 2003).

It has also been used, with more limited success, in support of causal analysis techniques such as Fault Tree Analysis (FTA). Some FTA approaches focus on automating construction of the fault tree from the design model (Papadopoulos and Maruhn 2001) while others focus on formal modelling of fault trees and verification of their correctness and completeness (Ortmeier and Schellhorn 2007). Typically the top event being analyzed is an undesirable system state or combination of component states, whereas our approach deals more generally with any hazard that can be expressed in LTL. (Rae and Lindsay 2004) use the FDR model checker to generate fault trees from process-algebraic models for hazards that are formalized in temporal logic. In (Lindsay et al. 2011) Behaviour Trees and model checking are used to support Cut Set Analysis (CSA) and automatically generate minimal cut sets from the model.

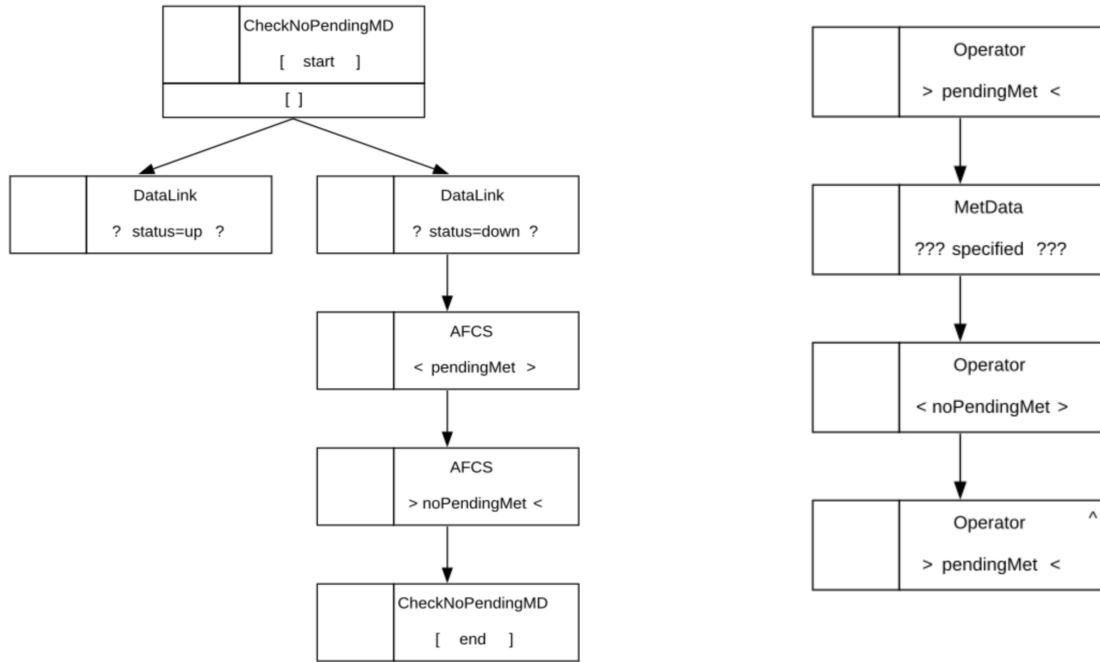

Figure 6: New safety feature SR-2: prompt operator to update pending met data

More recently, (Avrunin et al. 2010) outline a process modelling and analysis environment called Little-JIL and describe its application to improvement of healthcare processes. In (Simidchieva et al. 2010) they apply the approach to identify hazards in an election process. Their approach has many similarities to ours, including use of model checking to support FMEA and CSA. But development of models from natural languages specifications is not as straightforward as for Behaviour Trees.

10. SUMMARY AND CONCLUSIONS

Summary

In summary, the paper explores the use of Behaviour Trees and model checking to automate identification of system behaviours that can lead to hazardous system failures. A case study was used to illustrate the approach. The starting point was a set of 58 requirements for the AFMS system, taken directly from a much larger System-of-Systems Functional Performance Specification. The purely functional aspects of the requirements (as opposed to data requirements and performance aspects) were modeled in a Behaviour Tree. The BT model was extended with component failures, and conditions and events that could lead to unsafe outcomes, and system hazards were formulated within the new model. The SAL model checker was then used to identify system behaviours that could lead to the system hazards. System safety requirements were proposed to address some of these behaviours, and the method was reapplied to show that, indeed, the hazardous behaviours would no longer result.

Conclusions and Future Work

The case study illustrates that the BT method can successfully be extended to aid understanding of system technical risk early in the development life-cycle. Moreover, the models stay close to the natural language requirements specifications and the analysis is fully automated, which should enable the approach to be integrated more closely with standard systems engineering processes. This circumvents the criticisms often levelled at other formal methods and model-based systems engineering methods: namely, that they do not scale well and that tracing back to natural language specifications is difficult.

In future work we intend to extend the method to cover data relationships, which will broaden the range of technical risks that can be analysed. The first author is involved in another project where the

method is being applied to assist the development of a safety case for new HMI functionality in a safety-critical system, where the method will be used to check the effectiveness of safety control mechanisms. The third author is undertaking a PhD to investigate application of analysis techniques directly to BT models, rather than needing to translate to another formalism and use model checking to find a single counterexample at a time, in order to make analysis more efficient and to improve the utility of the analysis results.

ACKNOWLEDGEMENTS

This work was supported by Linkage Project grant LP0989363 from the Australian Research Council and Raytheon Australia. We gratefully acknowledge the guidance of Jim Boston and Dan Powell in the development and interpretation of the AFMS case study on which this work was based. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centres of Excellence program.

REFERENCES

Avrunin, G.S., Clarke, L.A., Osterweil, L.J. et al, "Experience Modelling and Analyzing Medical Processes: UMass/Baystate Medical Safety Project Overview", *Proc. 1st ACM Int. Health Informatics Symposium*, Arlington, Virginia, USA, 2010, pp. 316-325.

Bozzano, M. and Villafiorita, A. "Improving System Reliability Via Model Checking: The FSAP/Nusmv-SA Safety Analysis Platform". *Proc. Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*, LNCS vol. 2788, Springer-Verlag 2003, pp. 49–62.

Broy, M., "Can Practitioners Neglect Theory And Theoreticians Neglect Practice?", *IEEE Computer*, October 2011, pp. 19-24.

Cichocki, T., and Górski, J., "Formal Support For Fault Modelling And Analysis". *Proc. Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP)*, LNCS vol. 2187, Springer-Verlag 2001, pp. 190–199.

Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M., "NuSMV: A New Symbolic Model Verifier," *Proc. Int. Conf. on Computer Aided Verfication (CAV),* LNCS vol. 1633, Springer-Verlag, 1999, pp. 495–499.

Colvin, R. and Hayes, I. J., "A Semantics for Behaviour Trees Using CSP With Specification Commands", *Science of Computer Programming*, 76#10 (2011) 891-914.

de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M. and Tiwari, A., "SAL 2" *Proc. Int. Conf. on Computer-Aided Verification (CAV 2004)*, LNCS vol. 3114, Springer-Verlag, 2004, pp. 496–500.

Dromey, R.G, "From Requirements to Design: Formalizing the Key Steps", *Proc. 1st IEEE Int. Conference on Software Engineering and Formal Methods (SEFM)*, IEEE Computer Society, 2003, pp. 2-13.

Emerson, E.A., "Temporal and Modal Logic*", Handbook of Theoretical Computer Science, Volume B*, Elsevier Science Publishers, 1990, pp. 996-1072.

Grunske, L., Lindsay, P.A., Yatapanage, N. and Winter, K. "An Automated Failure Mode And Effect Analysis Based On High-Level Design Specification With Behaviour Trees". *Proc. Int Conf. on Integrated Formal Methods (IFM)*, LNCS vol. 3771, Springer-Verlag 2005, pp. 129–149.

Heimdahl, M.P.E., Choi, Y. and Whalen, M.W. "Deviation Analysis: A New Use Of Model Checking". *Automated Software Engineering*, 12#3 (2005) 321–347.

Leveson, N.G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

Lindsay P.A., "Behaviour Trees: From Systems Engineering To Software Engineering", *Proc. IEEE Int Conf on Software Eng. and Formal Methods (SEFM)*, Pisa, Italy, IEEE Computer Society,

2010, pp. 21–30.

Lindsay, P., Yatapanage, N. and Winter, K., “Cut Set Analysis Using Behaviour Trees And Model Checking” *Formal Aspects of Computing*, 2011, DOI: 10.1007/s00165-011-0181-8.

Papadopoulos, Y. and Maruhn, M. “Model-Based Synthesis Of Fault Trees From Matlab-Simulink Models”, *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2001)*, IEEE Computer Society, 2001, pp. 77-82.

Ortmeier, F. and Schellhorn, G. “Formal Fault Tree Analysis - Practical Experiences”. *Electronic Notes in Theoretical Computer Science*, 185, 2007, pp. 139 – 151

Powell, D. “Behaviour Engineering - A Scalable Modelling and Analysis Method”, *Proc. IEEE Int Conf on Software Eng. and Formal Methods (SEFM)*, Pisa, Italy, IEEE Computer Society, 2010, pp. 31-40.

Rae A., and Lindsay. P.. “A Behaviour-Based Method For Fault Tree Generation”. *Proc. Int. System Safety Conference,* System Safety Society, 2004, pp. 289-298.

Reese, J. D. and Leveson, N. G. “Software Deviation Analysis”. *Proc. 19th Int. Conf. on Software Engineering (ICSE)*, ACM Press, 1997, pp. 250–261.

Simidchieva, B.I., Engle, S.J. at el, “Modelling And Analyzing Faults To Improve Election Process Robustness”, *Proc. 2010 Int. Conf. on Electronic Voting Technology*, Washington, DC, USA, 2010, pp. 1-8.

BIOGRAPHIES

Dr Peter Lindsay is Boeing Professor of Systems Engineering at The University of Queensland and conducts many of his research projects within NICTA. He has held academic and research positions at the University of NSW, the University of Manchester and the University of Illinois at Urbana-Champaign. He is co-author of two books on formal specification and verification of software systems. He has worked on safety and security-critical applications in areas such as air traffic control, embedded medical devices, ship-board defence, emergency service dispatch systems, and an international diplomatic network. His research interests include engineering of complex systems, safety-critical systems, air traffic management, and formal methods of system development.

Dr Kirsten Winter is a research fellow at The University of Queensland. She has graduated in Computer Science and received her PhD in Germany before coming to Queensland. Her main research focus is on tool-supported analysis of software and systems, in particular fully automated techniques such as model checking. She has applied these techniques in a number of critical system applications, including automated verification of railway interlocking systems and tool-supported safety analyses.

Sentot Kromodimoeljo is a PhD student at The University of Queensland currently doing research in model checking. Prior to becoming a PhD student, he was a member of the EVES research and development group at IP Sharp Associates/Reuters and later at ORA Canada. He was co-architect of the EVES system and one of the main developers of the EVES theorem proving system. In addition to tool development, his work included formalisation and verification of military messaging systems and cryptographic protocols. His research interests include model checking, program analysis, theorem proving and cryptography.