Integrating safety and formal analyses using UML and PFS

Frantz Iwu *, Andy Galloway, John McDermid, Ian Toyn

Department of Computer Science, University of York, Heslington, York YO10 5DD, UK

Received 12 January 2005; received in revised form 7 November 2005; accepted 18 November 2005

Abstract

Where software systems are safety critical, for example in aircraft engine control, it is necessary to carry out safety analysis on designs in support of certification. We argue that there is also significant value in formally validating such a design. Few "classical" formal notations and methods are geared towards embedded systems. We illustrate one such method known as Practical Formal Specification (PFS), showing how it can be integrated in a UML context with various forms of safety analysis. The PFS method was developed to extend classical approaches in the development of embedded software systems in a way that adds engineering value, and fits into existing well-established frameworks. We exemplify the approach to model the reverse thrust selection function of the thrust reversal system of a turbo-jet engine.

© 2006 Elsevier Ltd. All rights reserved.

Keywords: Embedded systems; Formal methods; UML; Safety analysis

1. Introduction

The application of formal methods to the development of safety-critical software has been advocated by a number of standards (e.g. [1]), and mandated by at least one [2]. However, whilst there has been some work on the application of formal methods in various industrial and academic pilot projects, their use remains the exception, not the rule. There are many challenges, both practical and technical in nature, in the application of formal methods within this domain [3]. From our experience, one of the clearest reasons for this is that formal techniques are often treated as an isolated activity—relatively little emphasis is placed on how they integrate into the safety-critical system development process. In order to realise the practical benefits of formal methods and adhere to the spirit of standards, researchers in method integration are proposing ways to incorporate formalism into existing processes.

There are very few successful examples of the use of formal techniques ([4] is one), especially alongside other methodologies that are in use within an industrial development context. Our approach in this paper is to start from an existing and accepted conventional design method, the Unified Modelling Language (UML) [5], and sketch an approach that sets out to integrate safety and formal analyses (Fig. 1).

UML has become the de facto standard for object-oriented modelling of software systems. It uses various graphical notations to capture system requirements and to perform system analysis and design. Using UML, requirements are expressed using use-cases, concept diagrams, etc., and designs are expressed using sequence diagrams, class diagrams, etc. Such models are amenable to a number of forms of analyses, for example analysis that checks that a diagram is structurally well-formed. However, despite its widespread use, one of the drawbacks with UML is that it lacks formal semantics. Despite efforts being made to address this drawback, e.g. work on pUML [6], further work is required to address some of the important issues faced by developers of embedded safety-critical systems. The “disconnect” between the real world domain and the model, as well as between the model and the
implementation, are particular concerns. However, one of the benefits of UML is that it allows different stakeholders to develop certain aspects of software models independently. This ‘separation of concerns’ can be readily exploited by independently applying formal techniques to the different models employed during the software design process. Note, however, that some formal notion of consistency must be maintained between the different models for this exercise to be of most value.

Safety-critical systems are often complex and multi-faceted—they can simultaneously involve issues such as time, data integrity, communication constraints, etc. We attempt to separate some of these issues by decomposing the system design into three model views: architectural model, dynamic model and functional model as shown in Fig. 1. Each view represents different facets of the system and each is constructed and validated to ensure consistency. The dynamic and functional models describe the behaviour of individual components. The former describes the interactions with other components and the timing of these interactions. The latter comprises data definition and the data invariant, which include a component’s local state and its input/output relationships. The architectural model describes the relationships between the components used in the system. We adopt both the UML and Practical Formal Specification (PFS) methods in the definition of the architectural model.

One of the aims of this paper will be to illustrate the benefits of using the PFS method in modelling the discrete aspects of software used in control systems [7]. The PFS project [7–10] arose out of a desire to apply formal techniques to the development of software systems in a way that augmented the established development practices rather than replacing them. The tool support for PFS builds on the Matlab/Simulink/Stateflow toolset, which is used widely in control system development. This is consistent with other proposed approaches, e.g. by Nancy Leveson et al. for safety-critical control domains [11], and more generally by other researchers in method integration [12]. The PFS project has been running for nearly 9 years. The primary financial support has been from the UK Ministry of Defence (MoD). There have also been important industrial contributions to the programme, principally from Rolls-Royce and BAE SYSTEMS.

In the early stages of the PFS project, several case studies were undertaken, as “pencil and paper” exercises. These were inevitably limited in scope but still managed to find problems in some existing informal requirements, which had escaped reviews and testing. This work confirmed the benefits of early analysis, and provided the stimulus for the current phase of the project, which is primarily concerned with providing prototype tool support for the method [13].

PFS is useful for capturing essential requirements, which include “ideal” behaviour, and then later addressing “real-world” considerations such as component failures or noise on signals. One can also capture explicit assumptions on requirements which reflect the nature of the embedding system—this permits isolation between the model and its environment and provides a sound basis for abstraction [14]. The PFS method does not address all of the issues necessary to develop safety-critical software, but it does constitute a key link between the design and safety processes. It provides an effective medium for validating design information against (formally expressed) safety requirements.

Fig. 1. Safety and formal analyses framework.
Another aim of the paper is to show how parts of the three model views may be used as a basis for safety analyses. We especially advocate performing safety analysis as a basis for generating safety properties. These can then be expressed formally as PFS assumptions—either as pre- and post-conditions of a PFS “contract” or as assumptions on the root state of a PFS state machine. Safety properties expressed in this way can then be employed in the formal validation of the design by the Simulink/Stateflow Analyser (SSA) tool, which is being developed to support the PFS method. It is in this way that we achieve the integration of UML, safety analysis and formal methods.

2. Design approach: UML and PFS

This section introduces the methodology and the notation which are central to the paper. The method is illustrated through the analysis and design of an aircraft engine thrust reversal system, which uses “bucket doors” and is based on a partial set of requirements. The methodology used in the example combines both UML and PFS. UML provides a general architectural framework for the software system, while PFS facilitates the introduction of mathematical detail and the formal analysis of the resulting model.

2.1. Architectural model

The architectural model comprises use-cases, class diagrams and sequence diagrams. These techniques allow important aspects of a system to be modelled from different perspectives and at different levels of abstraction for example:

- **Use-cases:** These describe the required functionality of a system from a user’s perspective. For each case, this includes specifying the interactions between a user and the system in order to achieve a particular goal. Ultimately, all the system functions and use-cases should be traceable from requirements through to implementation and testing. In identifying use-cases, **scenarios** are useful. They describe a sequence of events, actions and transactions that may occur to achieve a use-case goal.

- **Class diagrams:** To cope with complexity, it is common to decompose the design, e.g. by divide and conquer. This involves dividing a problem domain space into comprehensible sub-domains. The quintessential step in object-oriented analysis is the decomposition of a problem domain into individual concepts or objects and showing the associations between them. Class diagrams are used in modelling the static structure of a system and describing the concepts or objects in a system and the relationships between them.

- **Sequence diagrams:** A scenario describes a sequence of events required to show how one instance of a use-case can achieve its goal. Scenarios, refining those identified in the use-cases, are expressed in UML using sequence diagrams. A sequence diagram shows the interaction between actors and objects in a system over time.

2.2. Detailed design framework

The detailed design framework permits the capture of essential requirements and state-based requirements, and also allows the identification of assumptions on requirements. These are expressed used PFS.

2.2.1. Capturing essential requirements

Often development starts by assuming an ideal environment, or context, for the system. Real-world considerations, such as noise on signals and environmental failures, are deferred until later in the evolution of a model.

By essential requirements, we mean behaviour in an ideal environment. Essential requirements are validated with respect to explicitly documented sets of assumptions upon which they rely. Later, as the ideal requirements are refined toward an actual design, assumptions may be “weakened” to accommodate real-world considerations, e.g. the rate of change assumptions on a particular input signal may be violated when sensors fail. When this happens, new functionality must be added to the requirements (and the system) to handle the additional behaviour admitted by the weakened assumptions. This is, of course, part of standard development practice, but PFS makes this explicit and analysable by means of refinement of assumptions, etc.

2.2.2. Capturing state-based requirements

When capturing requirements using PFS, it is usual to identify state-based components, which are described using hierarchical state-machines. Other components, such as those that abstractly characterise control law requirements, may be expressed using PFS contracts. However, this paper will primarily concern itself with state-based components—only a simple example of a contract is shown; for more information about PFS contracts, see [7].

State-based requirements describe how a component’s behaviour changes over the history of its inputs. The features of the PFS method described in this paper focus on synchronous systems, i.e. systems where components of software could be viewed as being clocked at a particular frequency and proceeding in lockstep. The synchronous part of the method allows certain requirements to be described by hierarchical state-machines. The PFS state-machine notation was inspired by Harel State Charts (and similar) but adopts a simpler syntax and semantics tailored towards explicit specification. It can be seen as a subset of the Stateflow and State Chart models.

The syntax adopted by PFS does not support orthogonality, i.e. AND states are precluded. However, parallel composition of state-machines can be achieved by
describing two or more components as state-machines at the architectural level. Also, events are precluded and communication is achieved only via the declared inputs and outputs of the component. Actions, for example assignments, are not allowed in transition labels.

The restrictions on PFS syntax make the semantics of the diagram highly explicit and force the specification to describe, in a clear way, what the system ought to do rather than how the system should achieve it. Some of these policies in PFS are currently under review (for example there is an issue of how to best address the problem of legacy systems specified using a richer notation). Another difference is that output signal definitions are associated with each state and not with transition actions. While the component remains in that state, the outputs are defined according to the definitions in that state. Each transition carries a trigger condition, which must hold true over the component’s input values in order for the transition to be taken. The unique initial transition that serves merely to identify the initial state is an exception.

Semantically, the state machine behaves synchronously. For every presentation of inputs, the machine evolves by taking one transition (if one is enabled), the outputs are set according to the definitions of the destination state (transition-then-compute semantics).

For example, consider the simple state-machine shown in Fig. 2. Here, the output definitions are shown explicitly in each child state: when in Child_State_A, Output_Y is set to Input_X-1, and when in Child_State_B, Output_Y is set to Input_X+1. The outputs are set according to the destination state, so if the state machine is in Child_State_A and an input (Input_X) arrives which satisfies the outgoing transition trigger to Child_State_B (i.e. it is greater than or equal to 10 and enable is true), then the machine moves into Child_State_B and the corresponding output (Output_Y) is set to Input_X+1. Otherwise, it remains in Child_State_A and the corresponding output (Output_Y) is set to Input_X-1.

2.2.3. Identifying assumptions on requirements

Assumptions are especially important for the purposes of abstraction and validating requirements. Explicitly identifying assumptions in the design of systems forms an essential part of the PFS process (indeed some of the errors we have found in analysing specifications reflect conflicts between specified behaviour and assumptions not stated). As a support for abstraction, the specifier need only specify behaviour within any stated assumptions. As an aid to confidence in validity, any model must be shown to respect assumptions wherever they are stated.

PFS state-machines permit several kinds of assumptions on state machines. Assumptions may be associated with both the leaf and compound states of a machine. Each sub-state inherits certain assumptions of its parent states. State assumptions validate outgoing transitions, whilst at the same time producing conformance criteria for incoming transitions. Assumptions may be used to provide an independent assurance of the validity of a state model. For example, if the assumptions on the source state of a transition are true then the transition condition must guarantee the assumptions of the target state.

During validation, nested assumptions must ultimately be shown to be correct with respect to the top-level assumptions for the whole state-machine (assumptions on the root super-state). This is achieved by showing that each state’s assumptions are correct with respect to their parent and each parent’s assumptions are correct with respect to their parent, etc. Certain assumptions may only be valid at certain times, e.g. in given phases of flight, and thus might only apply to a particular collection of states or super states.

We have outlined the nature of assumptions in PFS, but it is important to expand briefly upon how state assumptions aid abstraction. The root state of the machine may carry two forms of assumption on the inputs. First, the root state may carry assumptions about the valid range of inputs. Second, it may carry assumptions about the valid rate of change of the inputs. Note that the assumptions form a first-order pre-condition on the inputs (involving rate of change as well as range).

Each sub-state in the state-machine may also carry two forms of assumption. First, it may assume something about the last set of input values (which took the machine into that particular sub-state). Second, it may assume something about the way inputs change while in that state, i.e. about the next set of input values (which will possibly take the machine to a new state).

Ultimately, all sub-state assumptions can be proven contextually valid given the entering transitions to the sub-state, and the rate of change assumptions on the parent state. Crucially, the specifier must only provide exiting transitions for expected input values. Whilst this is asking the designer to do a little more than just specify what the software system does, this is information which the designer should understand; what PFS is doing here is providing a means of recording this information and using it to check the correctness of specifications.
2.2.4. Assumptions supported in PFS

PFS assumptions are mostly optional. However, without them, the automated analysis has only limited value. The role of assumptions was identified above. We now give a more complete description of the types of assumptions used in PFS state machines.

The types of assumptions associated with a PFS state include:

- Last assumptions (for any state, constraints on what the values of the inputs should have been for the state to become active).
- Initial assumptions (for a non-basic state, constraints on what the values of the inputs should have been for the state, and its initial state, to become active).
- Next assumptions (for any state, constraints on how the values of the inputs are expected to change while in the state).
- State preservation condition (for a basic state, under what conditions the state persists).

2.2.5. Tool support for PFS

The early stages of the PFS project were deliberately “neutral” with respect to the concrete syntax and engineering tools used for control system specification upon which PFS support could be based. However, for tool development it was necessary to choose a “host” tool as the basis for PFS analyses. Due to the widespread use and customisability of Matlab/Simulink/Stateflow (MSS) for control system specification, we chose to base the tool on this product. The tool supporting PFS is now called the Simulink/Stateflow Analyser (SSA).

SSA is integrated with MSS so that it appears as an extension to Simulink/Stateflow. New dialogues and facilities have been added. First, SSA customises Simulink/Stateflow to allow PFS assumptions to be recorded on models (our example focuses on the Stateflow elements). Second, it provides automated checks to ensure that assumptions are mutually consistent and that the model conforms to the assumptions. The SSA tool supports textual and tabular representations of assumptions. It is capable of organising constants and common definitions into lexicons (project dictionaries of definitions). It performs well-formedness checks on an annotated model to ensure that the model conforms to PFS syntactic conventions.

3. Safety analysis process

Software cannot directly cause harm but can cause failures, which could lead to hazards, through the systems it controls. The primary concern of system safety analysis is the management of hazards, but the complexity associated with software in this domain makes hazard management difficult.

Ideally, for safety-critical systems and software development processes to achieve safety in a top-down fashion, the process should include hazard identification, setting overall safety requirements and identifying and apportioning derived safety requirements (DSRs) to subsystems [15] as shown in Fig. 3. These processes highlight the importance of the two main elements of the preliminary system safety assessment (PSSA), as set out in the civil aerospace recommended practices ARP4761 [16]. A valid and complete identification of DSRs is a prerequisite to ensuring that the safety requirements on the system as a whole are met. Applying this approach initially involves capturing the intended behaviour of a system, identifying potentially faulty behaviour and thus determining hazards. The process continues via Functional Hazard Analysis (FHA) to identify ways in which software can contribute to hazards. This gives the basis for establishing DSRs on the software. Note: SSA in Fig. 3 means system safety analysis; this is not to be confused with our PFS support tool.
3.1. Safety analysis on UML models

The UML standard guidance [5] on the documentation of use-cases is limited beyond brief informal text. However, its use makes it suitable for representing functions of safety-critical systems. For example in:

- Defining system boundaries.
- Showing associations and relationships between actors (or sub-systems) and use-cases.
- Documenting general system functionality.

While use-cases are intended to represent general system functions and goals, scenarios describe a sequence of events, actions and transactions that may occur to achieve a use-case goal.

Several attempts have been made to adapt safety analysis techniques to UML. For example, the guidewords for Hazard and Operability Studies (HAZOP) defined in Defence Standard 00-58 [17] have been adapted to different UML model views [18]. Each element (use-cases, scenarios, classes, attributes and interactions) in the set of views is subject to deviation. This involves examining each element for potential deviation from the intended behaviour based on the category of failure modes such as omission, commission, time and value. An extensive set of methods for analysing OO systems especially software, has recently been developed [19–21]. Details of this approach are outside the scope of the paper, but complement the work described here. Scenarios are used to capture intentional behaviour of a system and not exceptional behaviour, which indicate errors. However, in performing safety analysis of systems, it is commonplace to identify exceptional behaviour, through the use of hazard analysis techniques, and then to specify mitigating strategies.

There is little guidance available in the UML standard for identification of exceptional behaviour using scenario or use-case descriptions. Our approach builds on the ideas in [19–21] and involves identifying exceptional cases and using these to define DSRs for software.

We have made several references to the safety properties of systems. One of the aims of an associated project (MATISSE) was to integrate formal methods and safety analysis. In particular, it aimed to show how to derive safety requirements, e.g. through fault trees, in a form that they could be specified as PFS contracts and shown to hold using SSA. In many cases the DSRs are constraints over sequences of values, not on single inputs and outputs. These can be represented using PFS assumptions over consecutive inputs and outputs—although there may be cases where it is preferable to use a state machine to specify safe behaviour. Theoretical work exists to support such analysis [22] (e.g. on higher order formulae and state models). However, at present SSA has limited capability for some of these features, so further practical work is required to support these forms of analyses.

4. Case study

This section describes how, within the UML framework, the PFS method can be used to model the reverse thrust selection function of a thrust reversal system. A thrust reversal system, found on modern aircraft jet engines, contributes to aircraft braking. It is particularly valuable on wet and icy runways where the reduced friction between tyres and the runway mean that wheel-braking systems are less effective. When this occurs, thrust reversal is especially important to the deceleration of an aircraft.

A thrust reversal system reverses the direction of the exhaust gas stream thus using engine power to create a deceleration force. Some of the operating principles [23] are:

- For each engine there is a reverse thrust selection lever in the cockpit, which is mounted on the engine thrust control lever (throttle), used to select reverse thrust. The reverse thrust selection lever is normally positioned in the stowed position.
- The reverse thrust selection lever has four positions: stowed, idle, normal and emergency. The first of these corresponds to forward thrust, the second of these correspond to idle thrust and the other positions (normal and emergency) correspond to reverse thrust power demand.
- The reverse thrust selection lever cannot be moved to a reverse thrust position unless isolation valve is true (i.e. weight on wheels and engine conditions are suitable for reverse thrust).
- The reverser doors deploy when the selection lever is in the idle position and other conditions permit. The door returns to forward thrust position when the lever is returned to the stowed position.
- Once deployment of the doors is initiated, an interlock in the reverse thrust selection lever system prevents the application of reverse thrust (normal or emergency) until the doors have been fully deployed.
- If the doors fail to move into full deployment position, the control system prevents the engine from being set to high power (to prevent mechanical engine damage).
- Under various flight phases and failure conditions, a mechanical lock holds the doors in the forward thrust position (stowed position).
- The operation of the thrust reversal is signalled to the pilots in various ways, including by a set of lights located in the cockpit.

The bucket door system is a hydraulically actuated system, as shown in Fig. 4. When the pilot selects reverse thrust, the doors are actuated by a hydraulic ram. On each engine, the position of the bucket doors are kept in synchronisation through mechanical links referred to as “drive idlers”. Although based on a real system, the case study must be viewed as idealised for illustration.
For the control system to allow reverse thrust, two pre-conditions must be satisfied. First, there must be an indication that the aircraft is on the ground. Sensors on the main undercarriage usually indicate this, generating a “weight on wheels” signal, which is logically true once the aircraft has touched down. Second, the reverse thrust selection lever must be in the idle (as opposed to stowed, normal or emergency) position.

In what follows we will focus in particular on the requirements for validating a pilot demand for reverse thrust.

4.1. UML specification

In this section, a UML model of the thrust reversal system is presented. Our UML model comprises a use-case diagram, a class diagram and a sequence diagram. The core aircraft function modelled is deceleration.

4.1.1. Use-case diagram

The overall use-case diagram represents an overview of the system’s functionality and the interactions between the pilot and the aircraft system to achieve the goal of effectively reducing the aircraft’s landing run on both dry and slippery runways.

As shown in Fig. 5, the core aircraft function (deceleration) is specified using UML use-case diagrams, which depict a pilot, a set of use-cases (ellipses) representing the sub-functions within the deceleration function and the associations between the pilot and the use-cases (for simplicity, aerodynamic drag is omitted).

Our primary concern is in the reverse thrust use-case. Before reverse thrust can be activated, there must be weight on wheels and the thrust lever must be in idle position. Sink rates and wheel rotation should also be considered but such analysis is outside the scope of this paper.

4.1.2. Class diagram

In the class diagram shown in Fig. 6, classes of objects are identified as well as their structure and relationships. UML Class diagrams have been used to describe the structure of this system. The structure consists of an Engine Frame, Bucket Doors, Hydraulic Rams, an Actuator and a Drive Idler.

4.1.3. Sequence diagram

In a sequence diagram, interactions between objects in a system are shown as well as information exchanged between objects. Scenarios corresponding to use-cases, which describe the sequence of events that may occur, are modelled using UML sequence diagrams. Interactions between objects in the system are usually described in terms of operations initiated by client objects. In sequence diagrams, time is of the essence and is represented as a vertical line. This depicts the lifetime of any object.

The UML sequence diagram in Fig. 7 describes a reverse thrust use-case, as introduced by the use-case diagram in Fig. 5.

4.1.4. State based models

PFS state machines differ from UML statecharts and Stateflow in that they carry assumptions not usually associated with UML statecharts or Stateflow. In modeling the state-based behaviour of a system using PFS, we need to present the hierarchical states and essential transitions of the system, with outputs\(^1\) defined per state. We then need to add information to the state model describing the assumptions made in each state and the conditions under which each state is preserved. This method provides the context in which formal reasoning can be applied to the requirements to ensure that the assumptions are mutually consistent and respected by the model. This is discussed further in Section 4.3.3.

4.2. Safety analysis

The safety analysis of the UML use-case is given in Table 1. From the model we identify actor’s intent (reverse thrust selection, decelerate aircraft on landing within required distance), pre- and post-conditions and event sequences.

Each element (pre-conditions, main events, post-conditions) of the use-case named “Decelerate Aircraft” shown in Table 1 is examined for potential deviation from the intended behaviour based on potential failure modes such

\(^1\) Output definitions for each basic state describe the relationship between the outputs and inputs while in that state.
as omission, commission, timing and value. Some of the results of this failure analysis are documented in Table 2.

Having identified all the hazards, it is necessary to perform Fault Tree Analysis (FTA) to show how credible faults combine to give rise to hazards. For a particular hazard, FTA aims to identify all potential causes which may lead to that hazard. Informally, a hazard is “an accident waiting to happen” and it is caused by faults or failures which occur in a particular context.

The fault tree shown in Fig. 8 is constructed from system information collected from the UML models as interpreted given an understanding of the domain. The fault tree shows how the top-level hazard “Reverse Thrust Deployed in Flight” could occur by identifying potential causes. Some
of the undeveloped events shown in the fault tree could be associated with the class and sequence models of the system.

For example, the “Thrust Controller Failed” event can be related to the Actuator class in the class model and “On Ground Sensing Failed” can be related to the SignalOnGround() event in the sequence model. Hence, information from the fault tree can be used to identify classes and interactions in the model in which more attention should be given; specifically these can be used to establish DSRs on specific elements of the control system.

4.3. PFS specification

To illustrate the approach adopted by PFS, involving state-machines and assumptions, we consider part of the thrust reversal selection system. In particular, we focus on the requirements for validating a demand for thrust reverse, and also the associated signal from the control system to the engine, which indicate whether increased reverse thrust is permitted. Fig. 9(a) shows the top-level window for a PFS specification consisting of just a state-machine component. It features the main elements of the SSA (Simulink/Stateflow Analyser) toolbox library. This provides the Simulink blocks needed for PFS assumptions to be recorded on the Simulink model.

The inputs to the state machine are as follows:

- **Lever**—The discrete input from the pilot for selecting reverse thrust. We represent stowed, idle thrust, normal reverse thrust and emergency reverse thrust as lever selections 1, 2, 3 and 4, respectively.\(^2\)

- **Isolation**—The discrete input which (when logically true) indicates that the isolation valve is open to permit reverse thrust activation. This can only occur when weight on wheels is logically true.

The output from the diagram is:

- **rt**—The discrete output that (when logically true) indicates that the model permits the application of increased engine thrust for reverse thrust.

Fig. 9(b) presents a hierarchical state-machine for a partial behaviour of the system. The states of the model are shown as labelled bubbles, with transition arrows connecting the states. The trigger conditions, which must hold true over the input values for the transitions to be taken, are shown near to each transition. In PFS, transitions should be taken on every presentation of inputs (or the state should persist under the conditions defined explicitly by the state preservation condition). SSA analysis ensures that the transition trigger conditions and state preservation condition cover every possible situation admitted by the state’s assumptions about the last and next set of inputs. Analysis also ensures they are pair-wise disjoint, guaranteeing deterministic behaviour.

As shown in Fig. 9(b), the diagram is hierarchical with one root super-state TRSystem encapsulating two sub-states RTOff and RTOn and a collection of basic states. TRSystem toggles between the RTOff and RTOn states. Transitions from their sub-states define how this occurs. A mechanical interlock is assumed to ensure that the lever cannot advance to position 3 until the Boolean input isolation has become true. When this occurs, the model evolves from the Idle state of RTOff into the Normal state of RTOn.

If the model is in the Normal state of RTOn then, depending on which lever position is selected next, the model evolves into the Idle state of RTOff or into the Emergency state.

The output definition for rt is false on entry into the Idle and Stowed states and while the model persists in these states. The output definition for rt is true on entry into the Normal and Emergency states and while the model persists in these states.

### 4.3.1. State preservation conditions

Having described the circumstances under which a change of state is required, we now define **state preservation conditions** under which no change of state is required. By complementing the information in the diagram with the state preservation conditions, the requirements can be validated for completeness and consistency. This ensures that every scenario concerning the next input values, within a state’s assumptions, is considered and resolved uniquely.

---

\(^2\)Normally a throttle setting of about 75% engine low-pressure shaft speed is used during normal reverse operation and 100% engine low-pressure shaft speed in an emergency, such as a high-speed rejected takeoff.
Table 2
Failures identification from system level analysis

<table>
<thead>
<tr>
<th>Element</th>
<th>Guide word</th>
<th>Deviation</th>
<th>Possible causes</th>
<th>Consequences</th>
<th>Safety requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft on ground (Pre-condition)</td>
<td>Commission</td>
<td>On ground detected when not true</td>
<td>Invalid airframe data; data transmission failure; system failure</td>
<td>Reverse thrust deployed in flight</td>
<td>Disallow thrust reverse when aircraft is not on ground; add interlock; provide auto restow</td>
</tr>
<tr>
<td>Aircraft on ground (Pre-condition)</td>
<td>Late</td>
<td>On ground not detected on time</td>
<td>Invalid airframe data; data transmission failure; system failure</td>
<td>Threat reverse deployed late</td>
<td>Redundancy on WoW sensors, data transmission medium</td>
</tr>
<tr>
<td>Reverse thrust selected (Main event)</td>
<td>Omission</td>
<td>Reverse thrust selection not detected when true</td>
<td>Invalid airframe data; data transmission failure; system failure</td>
<td>Threat reverse not deployed when required</td>
<td></td>
</tr>
<tr>
<td>Aircraft engine opened to low power setting (Main event)</td>
<td>Value</td>
<td>Aircraft engine power setting detected as high</td>
<td>Invalid airframe data; data transmission failure; system failure</td>
<td>Thrust reverse lever cannot be moved to reverse thrust position</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8. Simplified fault tree for reverse thrust deployment in flight.
The state preservation condition for *Idle* state is as follows:

**Idle, state preservation condition is**

\[ \text{lever} = 2 \]

Each set of inputs admitted by the next assumption for this state should enable exactly one of the outgoing transitions or make the state preservation condition true.

The state preservation condition for *Stowed* state is as follows:

**Stowed, state preservation condition is**

\[ \text{lever} = 1 \]

### 4.3.2. Adding state assumptions

Next we describe the assumptions for each state. In what follows we limit ourselves to examples from *RTOff* and its sub-states, and from the root state, TRSystem.

In the *Idle* state, we record two assumptions for the lever position: the last assumption and next assumption. The last assumption for the lever position will be guaranteed by the transitions in the model.

**Idle, last assumption is**

\[ \text{lever} = 2 \]

Considering the next possible set of values, we assume that the valid range of lever position is greater than or equal to 1 and less than or equal to 3, and if the next lever selection is equal to 3 then *isolation* also has to be true. The next assumption for *Idle* state is as follows:

**Idle, next assumption is**

\[ 1 < \text{lever} < 3 \quad \& \quad (\text{lever} = 3 \Rightarrow \text{isolation}) \]
These assumptions are guaranteed by the root state’s next assumptions in the context of Idle’s last assumption. In particular, the root state carries assumptions about the way the lever position is expected to change from one step to the next, as well as about the lever’s mechanical interlock.

In the Stowed state, we record two assumptions for the lever position: the last assumption and the next assumption. The last assumption for the lever position will be guaranteed by the transitions in the model.

Stowed, last assumption is

\[ \text{lever} \equiv 1 \]

Considering the next possible set of values, we assume that the valid range of lever position is greater than or equal to 1 and less than or equal to 2. The next assumption for Stowed state is as follows:

Stowed, next assumption is

\[ 1 < \text{lever} < 2 \]

This assumption is guaranteed by the root state’s next assumptions in context of the Stowed state’s last assumption. The root state carries assumptions about the way the lever position is expected to change from one step to the next.

Considering the initial assumptions for RTOff sub-state, we assume that the lever position is equal to 1 and isolation is false:

RTOff, initial assumption is

\[ \text{lever} \equiv 1 \land \neg \text{isolation} \]

The top-level assumptions (on the root state) form the basis upon which the correctness of the whole state-machine is founded. From the software perspective, these are genuine assumptions; they are properties on which the correct operation of the software depends but which the software cannot control. The thrust reverse selection system has four top-level next assumptions: the thrust reverse lever has to be in position 1, 2, 3 or 4; the thrust reverse lever will only move up or down by one position between presentations of inputs to the state machine. If the isolation valve is false the thrust reverse lever cannot be moved into normal or emergency positions—a mechanical interlock justifies this property. Finally, we assume a normal landing scenario in which isolation remains true during deployment of thrust reversers.

TRSystem, next assumption is

\[ 1 < \text{lever} < 4 \land \]

\[ (\text{lever-lever} \equiv 1 \land \text{lever} \equiv \text{lever-lever} \equiv 1) \land \]

\[ (\text{lever} \equiv 2 \land \text{lever} \equiv 3 \Rightarrow \text{isolation}) \land \]

\[ (\text{lever} \equiv 3 \land \text{lever} \equiv 3 \Rightarrow \text{isolation} \equiv \neg \text{isolation}) \]

TRSystem, initial assumption is

\[ \text{lever} \equiv 1 \land \neg \text{isolation} \]

Given a set of top-level assumptions, we can then propagate these, via the other components in the system, into conditions on the environment. These are then subject to other forms of validation. Subsequently, assumptions may be weakened systematically toward realistic real-world expectations and additional detail introduced in a controlled way.

4.3.3. Internal consistency of state models

Having recorded assumptions on a model, the SSA tool generates healthiness conditions to check that the model conforms to the assumptions. A typical healthiness condition, if established, ensures that a particular assumption is guaranteed by context.

The context that an assumption is placed in may be subject to parent assumptions. If this is the case, these are taken into account when generating the proof obligations associated with the assumption. SSA attempts to discharge healthiness conditions automatically using pre-defined tactics that exploit a variety of decision procedures, such as ‘sup-inf’ for goals involving linear arithmetic, simulated annealing for goals involving non-linear arithmetic, and model checking for goals involving finite domains. Whenever this automatic tactic fails, SSA uses another tactic to search for a counter-example. If that also fails, the access provided by SSA to the prover’s interactive mode can allow proofs to be found manually with the assistance of the prover; the incomplete proof produced by the first tactic can be perused.

Two of the healthiness conditions successfully proven by SSA are shown in Figs. 10 and 11.

For SSA state machines, the healthiness conditions that can be analysed given the necessary annotations include:

- **Next assumptions established**: that a state’s next assumptions are established (logically implied) by the combination of its last assumptions and its parent state’s next assumptions, see Fig. 10.
- **Last assumptions established**: that a transition’s trigger in combination with its source state’s last and next assumptions establishes its destination state’s last assumptions.
- **Last assumptions preserved**: that a state’s preservation condition in combination with its previous last assumption and its next assumption preserves the state’s last assumptions, see Fig. 11.
- **Initial assumptions established**: that a transition’s trigger in combination with its source state’s last and next assumptions establishes its destination state’s initial assumption.
- **Exiting transitions complete**: that the state’s preservation condition and exiting transitions cover all situations admitted by the state’s assumptions.
- **Exiting transitions disjoint**: that the state’s preservation condition and exiting transitions are pairwise disjoint, in other words specified behaviour is deterministic, within the state’s assumptions.
Fig. 10. Next assumption established for idle state.

Fig. 11. Last assumptions preserved for idle state.
4.3.4. Identification of derived safety requirements

There are two classes of safety requirement: fundamental safety requirements and DSRs. The former describes a set of top-level requirements on a project as set by standards or accepted engineering practice; the latter describes those requirements which involve detecting and mitigating hazards, or contributory causes, identified at all the stages of the safety lifecycle.

ARP4751 recommends that, within a given development phase, DSRs should be treated consistently with other requirements.

DSRs identify specific requirements to be met by components and the system design; they may also set out software safety integrity levels (SILs) or development assurance levels (DALs) for system component and safety processes and procedures. Here we focus on DSRs which affect system behaviour.

Some DSRs were identified for a number of failure conditions associated with this example; some of these are shown in Table 2. The following requirements were identified by a range of safety analyses.

- Disallow thrust reverser deployment when aircraft is not on ground.
- Implement interlock system to mitigate against unintentional thrust reverser deployment.
- Implement functionality that will automatically re-stow any inadvertent thrust reverser deployment.
- Detect delayed response time, which could potentially inhibit thrust reverser deployment and brake application.
- Allow for manual thrust reverser deployment to mitigate against failure by omission.
- Disallow thrust reverser deployment when engine conditions are not suitable.

The DSRs identified for this example include both functional and non-functional requirements. Some of these requirements are likely to be implemented as preconditions, which determine whether or not a use-case in question is allowed. The remaining requirements could be used to define additional use-cases that extend the decelerate aircraft function use-case, and which would need to be subject to hazard analysis.

4.3.5. Specifying safety contracts

A contract describes what a sub-system expects of its environment and commits to achieve. It emphasises what should happen, rather than how it should be achieved. Contracts are usually expressed in terms of pre- and post-conditions on operations. SSA can record these as annotations on models.

Having identified the DSRs for this example, it is important to specify them formally. There are various documented techniques such as Douglass’ real-time annotations [24], Object Constraint Language (OCL) [10], etc. suitable for specifying safety contracts on operations. The SSA tool extends the notation of Stateflow Conditions to allow reference to preceeding values of inputs and outputs (to capture differential information).

A SSA Contract block provides a dialogue for entering the pre- and post-conditions. In PFS pre- and post-conditions can be differentials, i.e. can refer to previous values of inputs and outputs. The properties that can be analysed given these annotations are:

- Contract satisfiable—that the domain of input–output relation (as described by the post condition) covers all cases allowed by the precondition.
- Contract satisfied—that the behaviour of the subsystem model conforms to the contract.

Having entered the pre- and post-conditions (and design), two healthiness check buttons are enabled thus allowing the analysis of the above properties to be performed. Fig. 12. An example contract block.
9(a) above shows our example (the Simulink diagram of Fig. 9(a)) with a SSA Contract block, and additional information. Opening that SSA Contract block and pressing the contract satisfiable button produces the dialogue shown in Fig. 12, which shows that the Contract is satisfiable.

5. Conclusions

In this paper we have illustrated an approach to developing safety-critical systems using the UML and PFS methods. The strategy of this approach requires that software analysis should start by systematically modelling the general architecture of software using UML, applying safety analysis techniques to UML models and performing detailed design involving the discrete behaviour of software using PFS.

In order to ensure mutual consistency between models and their associated assumptions, healthiness conditions are generated and proven by mathematical reasoning. To support this type of reasoning a tool SSA, which implements the PFS method, has been developed. It enables construction and maintenance of assumptions on state machines constructed in a MSS environment. In addition, the tool supports the generation and (generally automated) discharge of proof obligations from annotated models based on assistance from the CADiZ tool [13].

It may seem verbose to add assumptions to state transition diagrams. However, the system and software engineers developing the models will normally understand this information even if they never document it. Moreover, experience shows that it is often by making assumptions explicit that mistakes are identified. We would argue that for safety-critical systems the additional rigour is more than justified. PFS and SSA do not address all of the issues necessary to develop safety-critical software, but they provide a key link in mapping from the system and software safety process to (formal) software safety requirements.

The approach gives a basis for employing the rigour of formal methods to safety critical embedded systems. Furthermore, it does so in a way which is integrated with methods employed by practising software engineers. Thus, it is hoped that the method and tools provide a basis of a practical formal method for safety critical system development.

Acknowledgements

We acknowledge the support provided by the MoD under contract number FBG/01046 and the EPSRC-funded MATISSE project (Grant No. GR/R/70590/01).

References